

MPLAB® C18
C 编译器用户指南

请注意以下有关 Microchip 器件代码保护功能的要点：

- Microchip 的产品均达到 Microchip 数据手册中所述的技术指标。
- Microchip 确信：在正常使用的情况下， Microchip 系列产品是当今市场上同类产品中最安全的产品之一。
- 目前，仍存在着恶意、甚至是非法破坏代码保护功能的行为。就我们所知，所有这些行为都不是以 Microchip 数据手册中规定的操作规范来使用 Microchip 产品的。这样做的人极可能侵犯了知识产权。
- Microchip 愿与那些注重代码完整性的客户合作。
- Microchip 或任何其它半导体厂商均无法保证其代码的安全性。代码保护并不意味着我们保证产品是 “牢不可破”的。

代码保护功能处于持续发展中。 Microchip 承诺将不断改进产品的代码保护功能。任何试图破坏 Microchip 代码保护功能的行为均可视为违反了《数字器件千年版权法案（Digital Millennium Copyright Act）》。如果这种行为导致他人在未经授权的情况下，能访问您的软件或其它受版权保护的成果，您有权依据该法案提起诉讼，从而制止这种行为。

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip 位于美国亚利桑那州Chandler 和Tempe 及位于加利福尼亚州 Mountain View 的全球总部、设计中心和晶圆生产厂均于2003 年10 月通过了 ISO/TS-16949:2002 质量体系认证。公司在 PICmicro® 8 位单片机、 KEELOQ® 跳码器件、串行 EEPROM、单片机外设、非易失性存储器和模拟产品方面的质量体系流程均符合 ISO/TS-16949:2002。此外，Microchip 在开发系统的设计和生产方面的质量体系也已通过了 ISO 9001:2000 认证。



目录



前 言

简介

本文档论述 MPLAB® C18 编译器的技术细节，并讲解 MPLAB C18 编译器的所有功能。这里假定读者已经具备如下基本素质：

- 知道如何编写 C 程序
- 知道如何使用 MPLAB 集成开发环境创建和调试项目
- 已经阅读并理解了所使用单片机的数据手册

关于本指南

文档内容编排

文档内容编排如下：

- **第 1 章：简介** — 提供对 MPLAB C18 编译器的概述以及有关调用编译器的信息。
- **第 2 章：语法说明** — 论述 MPLAB C18 编译器与 ANSI 标准的不同之处。
- **第 3 章: 运行时模型** — 论述 MPLAB C18 编译器如何利用 PIC18 PICmicro® 单片机的资源。
- **第 4 章：优化** — 论述 MPLAB C18 编译器执行的优化功能。
- **第 5 章：示例应用程序** — 给出一个示例应用程序，并就本用户指南中论述的各主题，对源代码进行了说明。
- **附录 A：COFF 文件格式** — 详细阐述了 Microchip 的 COFF 格式。
- **附录 B: 采用 ANSI 定义的方式** — 论述按照 ANSI 标准的要求，MPLAB C18 实现所定义的执行方式。
- **附录 C：命令行概述** — 列出了命令行选项以及论述每个命令行选项的参考章节。
- **附录 D：MPLAB C18 诊断** — 列出了错误、警告和消息。
- **附录 E：扩展模式** — 论述非扩展模式和扩展模式之间的区别。

本指南中使用的约定

本用户指南使用如下文档约定：

文档约定

描述	表示	示例
代码 （**Courier** 字体）：		
Courier 字体	示例源代码	`distance -= time * speed;`
	文件名和路径	`c:\mcc18\h`
	关键字	`_asm, _endasm, static`
	命令行选项	`-Opa+, -Opa-`
斜体 Courier	可变参数	*`file`*`.o`，其中 *`file`* 可以是任何有效的文件名
方括号 [ ]	可选的参数	`mcc18 [options]` *`file`* `[options]`
省略号 ...	代替重复的文本示例	`var_name [, var_name...]`

	表示由用户提供的代码	`void main (void)`
		`{ ...`
		`}`

`0x`*`nnnn`*	十六进制数，其中 *n* 代表十六进制位	`0xFFFF, 0x007A`
文档 （**Arial** 字体）：		
斜体字符	参考书籍	*MPLAB User's Guide*

文档更新

所有的文档将来都会过时，本指南也不例外。为满足客户的需要， MPLAB C18 在不断发展之中，本文档中对于某些工具的描述可能与实际有所差别。请登录我公司网站获得最新的文档。

文档命名约定

文档用 "DS" 号编号。编号位于每页的页脚，在页码之前。 DS 号的命名约定为 DSXXXXXA 或 DSXXXXXA_CN，其中：

XXXXX = 文档号。
A = 文档的版本。
_CN = 文档为中文版。

PIC18 参考读物

readme.c18

关于使用 MPLAB C18 C 编译器的最新信息，请阅读本软件自带的 readme.c18 文件（ASCII 文本）。此自述文件包含了本文档可能未提供的更新信息。

MPLAB® C18 C 编译器入门 （DS51295C_CN）

描述如何安装 MPLAB C18 编译器，如何编写简单的程序以及如何使用安装了编译器的 MPLAB IDE 。

MPLAB® C18 C 编译器函数库 （DS51297C_CN）

关于 MPLAB C18 库文件和预编译目标文件的参考指南。列出了随 MPLAB C18 C 提供的所有库函数，并详细描述了这些库函数的使用。

MPLAB® IDE V6.XX 快速入门指南 （DS51281C_CN）

描述如何安装 MPLAB IDE 软件，如何使用它来创建项目及烧写器件。

MPASM™ User's Guide with MPLINK™ Linker and MPLIB™ Librarian (DS33014)

讲述如何使用 Microchip PICmicro MCU 汇编器 （MPASM）、链接器 （MPLINK）和库管理器 （MPLIB）。

PICmicro® 18C 单片机系列参考手册 （DS39500A_CN）

重点介绍增强型单片机系列。说明了增强型单片机系列的架构和外设模块的工作原理，但没有涉及到每个器件的具体细节。

PIC18 Device Data Sheets and Application Notes

讲述 PIC18 器件工作和电气特性的数据手册。应用笔记介绍了如何使用 PIC18 器件。

要获得上面列出的任何文档，请访问 Microchip 的网站 （www.microchip.com），获得 Adobe Acrobat （.pdf）格式的文档。

C 语言参考读物

American National Standard for Information Systems – *Programming Language – C*. American National Standards Institute (ANSI), 11 West 42nd. Street, New York, New York, 10036.

此标准规定了用 C 语言编写程序的格式，并对 C 程序进行了解释。其目的是提高 C 程序在多种计算机系统上的可移植性、可靠性、可维护性及执行效率。

Beatman, John B. *Embedded Design with the PIC18F452 Microcontroller*, First Edition. Pearson Education, Inc., Upper Saddle River, New Jersey 07458.

重点介绍 Microchip 公司的 PIC18FXXX 系列单片机以及如何编写优化的应用代码。

Harbison, Samuel P. and Steele, Guy L., *C A Reference Manual,* Fourth Edition. Prentice-Hall, Englewood Cliffs, New Jersey 07632.

详细地讲述了 C 编程语言。这本书是一本权威性的参考手册，它对 C 语言、运行时库以及 C 编程的风格都进行了完整的描述， C 编程强调正确性、可移植性和可维护性。

Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language*, Second Edition. Prentice Hall, Englewood Cliffs, New Jersey 07632.

对由 ANSI 标准定义的 C 语言进行了简明阐述。对于 C 程序员来说是一本出色的参考书。

Kochan, Steven G. *Programming In ANSI C*, Revised Edition. Hayden Books, Indianapolis, Indiana 46268.

学习 ANSI C 的另一本出色的参考书，用作大学教材。

Van Sickle, Ted. *Programming Microcontrollers in C*, First Edition. LLH Technology Publishing, Eagle Rock, Virginia 24085.

尽管这本书主要讲的是 Motorola 单片机，但其中单片机 C 语言编程的基本原理是很有用的。

MICROCHIP 网站

Microchip 网站为您提供在线支持。客户很容易从 Microchip 网站上获得文件和信息。要访问此网站，您必须能访问互联网并要安装

Netscape Navigator® 或 Microsoft® Internet Explorer 等网络浏览器。

使用您喜欢的 Internet 浏览器，可以访问 Microchip 的网站：

http://www.microchip.com

网站提供多种服务。用户从网站上可以下载到最新开发工具的文件、数据手册、应用笔记、用户指南、文章和示例程序。也可以获得关于 Microchip 业务的具体信息，包括销售办事处、分销商和工厂代表的列表。

技术支持

- 常见问题 （FAQ）
- 在线讨论组 — 关于产品、开发系统、技术信息及其它方面的讨论会。
- Microchip 顾问计划成员列表
- 到其它与 Microchip 产品相关的其它有用网站的链接

工程师工具箱

- 设计技巧
- 器件勘误表

其它信息

- 最新 Microchip 新闻稿
- 研讨会和活动列表
- 招聘职位

开发系统客户通知服务

Microchip 启动了客户通知服务，来帮助客户轻松获得关于 microchip 产品的最新信息。订阅此项服务后，每当您指定的产品系列或感兴趣的开发工具有更改、更新、改进或有勘误时，您都会收到电子邮件通知。

登录 Microchip 网站 （http://www.microchip.com），点击 “客户变更通知”。按照指示注册。

开发系统产品组分类如下：

- 编译器
- 仿真器
- 在线调试器
- MPLAB IDE
- 编程器

下面是对这些类别的描述：

编译器 — 关于 Microchip C 编译器和其它语言工具的最新信息。这些工具包括 MPLAB C17、 MPLAB C18 和 MPLAB C30 C 编译器； MPASM 和 MPLAB ASM30 汇编器； MPLINK 和 MPLAB LINK30 目标链接器； MPLIB 和 MPLAB LIB30 目标库管理器。

仿真器 — 关于 Microchip 在线仿真器的最新信息。包括 MPLAB ICE 2000 和 MPLAB ICE 4000。

在线调试器 — 关于 Microchip 在线调试器的最新信息，包括 MPLAB ICD 2。

MPLAB IDE — 关于 Microchip MPLAB IDE 的最新信息，它是开发系统工具的 Windows® 集成开发环境。重点介绍 MPLAB IDE、MPLAB SIM 仿真器、MPLAB IDE 项目管理器以及一般的编辑和调试功能。

编程器 — 关于 Microchip 器件编程器的最新信息。包括 MPLAB PM3 和 PRO MATE® II 器件编程器，以及 PICSTART® Plus 开发编程器。

客户支持

Microchip 产品的用户可通过下列渠道获得支持：

- 分销商或代表
- 当地销售办事处
- 应用工程师 （FAE）
- 应用工程师 （CAE）
- 热线

客户可以致电其分销商、代表或应用工程师 （FAE）寻求支持。请查看本手册后封面的销售办事处及地址列表。

欲获得技术支持，可访问网站 http://support.microchip.com，也可致电应用工程师（CAE），中国大陆地区请拨打 800-820-6247。

此外还有系统信息和更新热线。此热线为系统用户提供开发系统软件产品最新版本的列表。此热线还提供有关客户如何获得目前更新工具包的信息。

热线号码为：

美国和加拿大大部分地区，请拨打 1-800-755-2345。

全球其它国家或地区，请拨打 1-480-792-7302。



第 1 章 简介

1.1 概述

MPLAB C18 编译器是适用于 PIC18 PICmicro 单片机的独立而优化的 ANSI C 编译器。仅在 ANSI 标准 X3.159-1989 与高效的 PICmicro 单片机支持有冲突的情况下，此编译器才会与 ANSI 标准有所偏离。此编译器是一个 32 位 Windows 平台应用程序，与 Microchip 的 MPLAB IDE 完全兼容，它允许使用 MPLAB ICE 在线仿真器、 MPLAB ICD 2 在线调试器或 MPLAB SIM 软件模拟器进行源代码级调试。

MPLAB C18 编译器有以下特点：

- 与 ANSI '89 兼容
- 能集成到 MPLAB IDE，便于进行项目管理和源代码级调试
- 能生成可重定位的目标模块，增强代码的重用性
- 与由 MPASM 汇编器生成的目标模块兼容，允许在同一个项目中自由地进行汇编语言和 C 语言的混合编程
- 对外部存储器的读 / 写访问是透明的
- 当需要进行实时控制时能很好地支持行内汇编
- 具有多级优化的高效代码生成引擎
- 拥有广泛的库支持，包括 PWM、SPI™、I^2C™、UART、USART、字符串操作和数学函数库
- 用户能对数据和代码的存储空间分配进行完全控制

1.2 调用编译器

《MPLAB® C18 C 编译器入门》（DS51295C_CN）描述了如何在 MPLAB IDE 中使用 C18 编译器。也可以通过命令行调用编译器，命令行用法如下：

```
mcc18 [options] file [options]
```

可以指定一个源文件和任意个命令行选项。`--help` 命令行选项列出编译器接受的所有命令行选项。`-verbose` 命令行选项使编译器在编译结束时显示版本号以及错误、警告和消息的总数等信息。

1.2.1 生成输出文件

默认情况下，编译器会生成一个名为 *file*.o 的输出目标文件，其中， *file* 是在命令行上指定的源文件名，不包括扩展名。可通过-fo 命令行选项改变输出目标文件名。例如：

```
mcc18 -fo bar.o foo.c
```

如果源文件有错误，那么编译器会生成一个名为 *file*.err 的错误文件，其中，*file* 是在命令行上指定的源文件名，不包括扩展名。可通过-fe 命令行选项改变错误文件名。例如：

```
mcc18 -fe bar.err foo.c
```

1.2.2 显示诊断信息

诊断信息可通过-w 和-nw 命令行选项控制。-w 命令行选项设置警告诊断的级别（1、2 或 3）。表 1-1 列出了警告诊断的级别以及所表示的诊断类型。-nw 命令行选项禁止特定的消息 （附录 D 或 --help-message-list 命令行选项列出由编译器生成的所有消息）。使用--help-message-all 命令行选项，可得到关于所有消息的帮助。若要获得关于某个特定诊断的帮助，可使用 --help-message 命令行选项。例如：

```
mcc18 --help-message=2068
```

会显示以下结果：

```
2068: obsolete use of implicit 'int' detected.

The ANSI standard allows a variable to be declared without a base type
being specified, e.g., "extern x;", in which case a base type of 'int'
is implied. This usage is deprecated by the standard as obsolete, and
therefore a diagnostic is issued to that effect.
```

表 1-1： 警告级别

警告级别	所表示的诊断
1	错误 （致命的和非致命的）
2	级别 1 加警告
3	级别 2 加消息

1.2.3 定义宏

-D 命令行选项允许定义宏。可以用如下两种方式之一指定-D 命令行选项：-D*name* 或 -D*name*=*value*。-D*name*定义宏名为*name*并设定其值为1；-D*name*=*value*定义宏名为 *name* 并设定其值为 *value*。例如：

```
mcc18 -DMODE
```

定义了宏 MODE，其值为 1，而：

```
mcc18 -DMODE=2
```

定义宏 MODE 的值为 2。

使用-D 命令行选项的一个例子是条件编译，例如：

```
#if MODE == 1
  x = 5;
#elif MODE == 2
  x = 6;
#else
  x = 7;
#endif
```

1.2.4 选择处理器

默认情况下， MPLAB C18 针对一般的 PIC18 PICmicro 单片机编译应用程序。可以利用-p*processor* 命令行选项指定为某个特定的处理器生成目标文件，其中 *processor* 指定要使用的处理器型号。例如，要生成仅供 PIC18F452 使用的目标文件，应该使用命令行选项-p18f452。命令行选项-p18cxx 明确指定针对一般的 PIC18 PICmicro 单片机编译源文件。

1.2.5 选择模式

编译器可工作在如下两种不同的工作模式：扩展模式和非扩展模式。工作在扩展模式时，编译器使用扩展指令 （即 ADDFSR、 ADDULNK、CALLW、MOVSF、MOVSS、PUSHL、 SUBFSR 和 SUBULNK）和立即数变址寻址，这种寻址方式通常需要较少的指令来访问基于堆栈的变量 （因此占用较小的程序存储空间）。工作在非扩展模式时，编译器不使用扩展指令或立即数变址寻址。--extended 和 --no-extended 命令行选项告知编译器工作模式。表 1-2 概括了基于所指定命令行选项的编译器工作模式。

表 **1-2**： 模式选择

	-p *extended*	**-p *no-extended***	**-p18cxx**	不指定编译器
`--extended`	扩展	错误	扩展	扩展
`--no-extended`	非扩展	非扩展	非扩展	非扩展
不指定	非扩展	非扩展	非扩展	非扩展

注： 如果使用 `mcc18 --help` 调用编译器，将显示关于编译器工作在非扩展模式的帮助；但是，当编译器工作在非扩展模式时，不是所有的命令行选项都有效。要查看关于编译器工作在扩展模式时的帮助，应该使用命令行选项 `mcc18 --extended --help`。

注： 其它命令行选项将在本用户指南的后面部分中论述，在附录 C 中可以找到对所有命令行选项的概括。



第 2 章 语法说明

2.1 数据类型及数值范围

2.1.1 整型

MPLAB C18 编译器支持由 ANSI 定义的标准整型。标准整型的数值范围如表 2-1 所示。另外， MPLAB C18 还支持 24 位整型 `short long int`（或 `long short int`），分为有符号和无符号两种类型。表 2-1 也列出了 24 位整型的数值范围。

表 **2-1**： 整型数据的长度及数值范围

类型	长度	最小值	最大值
`char`[1,2]	8 位	-128	127
`signed char`	8 位	-128	127
`unsigned char`	8 位	0	255
`int`	16 位	-32,768	32,767
`unsigned int`	16 位	0	65535
`short`	16 位	-32,768	32,767
`unsigned short`	16 位	0	65,535
`short long`	24 位	-8,388,608	8,388,607
`unsigned short long`	24 位	0	16,777,215
`long`	32 位	-2,147,483,648	2,147,483,647
`unsigned long`	32 位	0	4,294,967,295

注 **1**： 若 `char` 前没有符号说明，则默认为有符号型。
2： 可通过`-k` 命令行选项使无符号说明的 `char` 默认为无符号型。

2.1.2 浮点型

对 MPLAB C18 来说， `double` 或 `float` 数据类型都是 32 位浮点型。表 2-2 列出了浮点型数据的数值范围。

表 2-2： 浮点型数据的长度及数值范围

类型	长度	最小指数	最大指数	规格化的最小值	规格化的最大值
`float`	32 位	-126	128	$2^{-126} \approx 1.17549435e - 38$	$2^{128} * (2\text{-}2^{-15}) \approx 6.80564693e + 38$
`double`	32 位	-126	128	$2^{-126} \approx 1.17549435e - 38$	$2^{128} * (2\text{-}2^{-15}) \approx 6.80564693e + 38$

MPLAB C18 的浮点数格式是 IEEE 754 格式的改进形式。 MPLAB C18 格式和 IEEE 754 格式的不同之处在于数据表示的最高 9 位。 IEEE 754 格式的最高 9 位循环左移一次将转换为 MPLAB C18 格式。同理，MPLAB C18 格式最高 9 位循环右移一次将转换为 IEEE 754 格式。表 2-3 对这两种格式作了比较。

表 2-3： MPLAB C18 浮点格式与 IEEE 754 格式的对比

标准	指数字节	字节 0	字节 1	字节 2
IEEE 754	$se_0e_1e_2e_3e_4e_5e_6$	e_7*ddd dddd*	*dddd dddd*	*dddd dddd*
MPLAB C18	$e_0e_1e_2e_3e_4e_5e_6e_7$	*sddd dddd*	*dddd dddd*	*dddd dddd*

图注： s = 符号位
d = 尾数
e = 指数

2.2 字节存储顺序 — ENDIANNESS

Endianness 指多字节数据中的字节存储顺序。 MPLAB C18 采用低字节低地址（little-endian）格式存储数据，低字节存储在较低地址中 （即数据是按 “低字节先存” 的方式存储的）。例如：

```
#pragma idata test=0x0200
long l=0xAABBCCDD;
```

数据在存储器中的存放结果如下：

地址	***0x0200***	***0x0201***	***0x0202***	***0x0203***
内容	0xDD	0xCC	0xBB	0xAA

2.3 存储类别

MPLAB C18 支持 ANSI 标准的存储类别（auto、extern、register、static 和 typedef）。

2.3.1 Overlay

MPLAB C18 编译器引入了 overlay（重叠）存储类别，仅当编译器工作在非扩展模式（参见 **1.2.5 节 “选择模式”**）时才使用此存储类别。overlay 存储类别可用于局部变量（但不能用于形式参数、函数定义或全局变量）。overlay 存储类别将相关变量分配到一个特定于函数的静态重叠存储区。这种变量是静态分配存储空间的，但每次进入函数时都要被初始化。例如：

```
void f (void)
{
  overlay int x = 5;
  x++;
}
```

尽管 x 的存储空间是静态分配的，x 在每次进入函数时都会被初始化为 5。如果没有初始化，那么进入函数时其值是不确定的。

MPLINK 链接器将使不同时运行的函数中定义为 overlay 的局部变量共享存储空间。例如，在下面的函数中：

```
int f (void)
{
  overlay int x = 1;
  return x;
}

int g (void)
{
  overlay int y = 2;
  return y;
}
```

如果 f 和 g 永远不会同时运行，则 x 和 y 共享相同的存储空间。但是，在下面的函数中：

```
int f (void)
{
  overlay int x = 1;
  return x;
}

int g (void)
{
  overlay int y = 2;
  y = f ( );
  return y;
}
```

由于 f 和 g 可能会同时运行，x 和 y 不能共享相同的存储空间。使用 overlay 局部变量的优点是，其存储空间是静态分配的，也就是说，在一般情况下，存取这种变量所需要的指令较少（因此所生成代码占用的程序存储空间也较小）。同时，由于一些变量可以共享相同的存储空间，这些变量所需分配的总的数据存储空间比定义为 static 时要小。

如果 MPLINK 链接器检测到包含 overlay 局部变量的递归函数，就会发出错误并中止编译。如果 MPLINK 链接器检测到，在任意模块中有通过指针进行的函数调用，在任意模块 （不一定和上述模块是同一模块）中有存储类别为 overlay 的局部变量，就会发出错误并中止编译。

局部变量默认的存储类别是 auto。可以使用关键字 static 或 overlay 显式地定义存储类别，或使用-scs （static 局部变量）或-sco （overlay 局部变量）命令行选项隐式地定义存储类别。为保持完整性， MPLAB C18 也支持-sca 命令行选项，该选项允许把局部变量的存储类别显式地定义为 auto 型。

2.3.2 static 型函数参数

函数参数的存储类别可以是 auto 型或 static 型。 auto 型参数存放在软件堆栈中，允许重入。 static 型参数是全局分配存储空间的，允许直接访问，通常所需代码较少。 static 型参数仅当编译器工作在非扩展模式 （参见 **1.2.5 节 "选择模式"**）时有效。

函数参数默认的存储类别是 auto 型。可以使用关键字 static 显式地定义存储类别或使用-scs 命令行选项隐式地定义存储类别。-sco 命令行选项也可以隐式地把函数参数的存储类别改变为 static 型。

2.4 存储限定符

除 ANSI 标准的存储限定符 （const, volatile）外， MPLAB C18 编译器还引入了 far、near、 rom 和 ram 存储限定符。在语句构成上，这些新限定符与标识符之间的约束关系与 ANSI C 中 const 和 volatile 限定符与标识符的关系相同。表 2-4 表明，定义对象时所指定的存储限定符决定了对象在存储器中的位置。对于一个没有用显式的存储限定符定义的对象，其默认的存储限定符是 far 和 ram。

表 2-4： 存储限定符指定对象在存储器中的位置

	rom	ram
far	程序存储器中的任何位置	数据存储器中的任何位置 （默认）
near	在程序存储器中的地址小于 64K	存取存储区

2.4.1 near/far 数据存储对象

far 限定符表示变量存储在数据存储器的存储区中，访问这一变量之前需要存储区切换指令。 near 限定符表示变量存储在存取 RAM 中。

2.4.2 near/far 程序存储对象

far 限定符表示变量可以位于程序存储器中的任何位置，或者，如果是一个指针变量，那么它能访问 64K 或者更大的程序存储空间。 near 限定符表示变量只能位于地址小于 64K 的程序存储空间，或者，如果是一个指针变量，那么它只能访问不超过 64K 的程序存储空间。

2.4.3 ram/rom 限定符

因为 PICmicro 单片机使用独立的程序存储器和数据存储器地址总线，所以 MPLAB C18 需要一些扩展来区分数据是位于程序存储器还是位于数据存储器。ANSI/ISO C 标准允许代码和数据位于不同的地址空间，但并不能定位代码空间中的数据。为此，MPLAB C18 引入了 rom 和 ram 限定符。rom 限定符表示对象位于程序存储器中，而 ram 限定符表示对象位于数据存储器中。

指针既可以指向数据存储器 （ram 指针），也可以指向程序存储器 （rom 指针）。一般将指针视为 ram 指针，除非定义为 rom。指针的长度取决于指针的类型，如表 2-5 所示。

> **注：** 写 rom 变量时，编译器使用一个 TBLWT 指令；但可能还需要附加的应用代码，这取决于所使用的存储器类型。详情请参阅数据手册。

表 2-5： 指针长度

指针类型	例子	长度
数据存储器指针	`char * dmp;`	16 位
Near 程序存储器指针	`rom near char * npmp;`	16 位
Far 程序存储器指针	`rom far char * fpmp;`	24 位

2.5 包含文件搜索路径

2.5.1 系统头文件

在 MCC_INCLUDE 环境变量中指定的路径和由-I 命令行选项指定的目录中搜索用 #include <*filename*> 包含的源文件。MCC_INCLUDE 环境变量和-I 值都是由分号界定的搜索目录列表。如果被包含的文件在 MCC_INCLUDE 环境变量列出的目录和-I 命令行选项列出的目录中都存在，那么将从由-I 命令行选项列出的目录中包含此文件，而忽略 MCC_INCLUDE 环境变量列出的目录。

2.5.2 用户头文件

先在引用被包含文件的源文件所在目录中搜索用 #include "*filename*" 包含的源文件。如果找不到，则采用搜索系统头文件的方式搜索此文件 （参见 **2.5.1 节 "系统头文件"**）。

2.6 预定义宏名

除了标准的预定义宏名外， MPLAB C18 还提供了如下预定义宏：

__18CXX 常数 1，用来表明使用的是 MPLAB C18 编译器。

__PROCESSOR 如果是为某个处理器进行编译的话，则相应的值为常数 1。例如，如果是用-p18c452 命令行选项编译，那么 __18C452 为常数 1。

如果是用-p18f258 命令行选项编译，那么 __18F258 为常数 1。

__SMALL__ 若是用-ms 命令行选项编译，为常数 1。

__LARGE__ 若是用-ml 命令行选项编译，为常数 1。

__TRADITIONAL18__ 如果使用非扩展模式 （参见 **2.5.1 节 "系统头文件"**），为常数 1。

__EXTENDED18__ 如果使用扩展模式 （参见 **2.5.1 节 "系统头文件"**），为常数 1。

2.7 与 ISO 的差异

2.7.1 整型的提升

根据 ISO 的要求，所有算术运算都以 int 精度或更高精度进行。在默认情况下，MPLAB C18 的算术运算以最大操作数的长度进行，即使两个操作数长度都小于 int 也不例外。可通过-Oi 命令行选项设定按 ISO 标准进行运算。例如：

```
unsigned char a, b;
unsigned i;

a = b = 0x80;
i = a + b; /* ISO requires that i == 0x100, but in C18 i == 0 */
```

注意，常数也存在同样的差异。为常数选择的类型是所有合适的类型中长度最小的类型，所谓合适的类型指能无溢出地表示常数值的类型。

例如：

```
#define A 0x10 /* A will be considered a char unless -Oi
                  specified */
#define B 0x10 /* B will be considered a char unless -Oi
                  specified */
#define C (A) * (B)

unsigned i;
i = C; /* ISO requires that i == 0x100, but in C18 i == 0 */
```

2.7.2 数字常量

MPLAB C18 支持指定十六进制 （0x）和八进制 （0）值的标准前缀，另外还支持用前缀 0b 来指定二进制值。例如，数值 237 可以表示为二进制常数 0b11101101。

2.7.3 字符串常量

程序存储器中的数据主要是静态字符串。为此，MPLAB C18 自动把所有字符串常量存放在程序存储器中。这种类型的字符串常量是 “位于程序存储器的 char 数组” （const rom char []）。 .stringtable 段是一个包含所有常量字符串的 romdata （参见 **2.9.1 节 “#pragma sectiontype”**）段。例如，如下的字符串 “hello” 将被置于 .stringtable 段：

```
strcmppgm2ram (Foo, "hello");
```

由于常量字符串存放在程序存储器中，所以标准字符串处理函数有多种形式。例如， strcpy 函数就有四种形式，允许把数据存储器或程序存储器中的字符串拷贝到数据存储器或程序存储器：

```
/*
 * Copy string s2 in data memory to string s1 in data memory
 */
char *strcpy (auto char *s1, auto const char *s2);

/*
 * Copy string s2 in program memory to string s1 in data
 * memory
 */
char *strcpypgm2ram (auto char *s1, auto const rom char *s2);

/*
 * Copy string s2 in data memory to string s1 in program
 * memory
 */
rom char *strcpyram2pgm (auto rom char *s1, auto const char *s2);

/*
 * Copy string s2 in program memory to string s1 in program
 * memory
 */
rom char *strcpypgm2pgm (auto rom char *s1,
                         auto const rom char *s2);
```

当使用 MPLAB C18 时，程序存储器中的一个字符串表可以定义为：

```
rom const char table[][20] =   { "string 1", "string 2",
                                 "string 3", "string 4" };
rom const char *rom table2[] = { "string 1", "string 2",
                                 "string 3", "string 4" };
```

table 定义为一个由四个字符串组成的数组，每个字符串的长度为 20 个字符，所以在程序存储器中占据 80 个字节。 table2 定义为一个指向程序存储器的指针数组。 * 后面的 rom 限定符表示把指针数组也存放在程序存储器中。 table2 中的所有字符串长度均为 9 个字节，而数组有 4 个元素，所以 table2 在程序存储器中共占用了（9*4+4*2）= 44 个字节。然而，对 table2 的存取可能会比对 table 的存取效率要低，这是由于指针需要附加的间接寻址指令。

MPLAB C18 独立地址空间的一个重要影响是指向程序存储器中数据的指针与指向数据存储器中数据的指针不兼容。只有当两种指针指向兼容类型的对象，而且指向的对象位于相同的地址空间时，两种指针才会兼容。例如，一个指向程序存储器中字符串的指针与一个指向数据存储器中字符串的指针是不兼容的，因为它们指向不同的地址空间。

把一个字符串从程序存储器拷贝到数据存储器的函数可以这样编写：

```
void str2ram(static char *dest, static char rom *src)
{
  while ((*dest++ = *src++) != '\0')
    ;
}
```

下面的代码利用 PICmicro 单片机 C 库函数把一个位于程序存储器的字符串送到 PIC18C452 的 USART。库函数 `putsUSART(const char *str)` 用来把字符串送到 USART，它把指向一个字符串的指针作为其参数，但是此字符串必须位于数据存储器中。

```
rom char mystring[] = "Send me to the USART";

void foo( void )
{
  char strbuffer[21];
  str2ram (strbuffer, mystring);
  putsUSART (strbuffer);
}
```

另一种方法是，可以把库函数修改为从程序存储器中读字符串。

```
/*
 * The only changes required to the library routine are to
 * change the name so the new routine does not conflict with
 * the original routine and to add the rom qualifier to the
 * parameter.
 */
void putsUSART_rom( static const rom char *data )
{
  /* Send characters up to the null */
  do
  {
    while (BusyUSART())
      ;

    /* Write a byte to the USART */
    putcUSART (*data);
  } while (*data++);
}
```


2.8 语言的扩展

2.8.1 匿名结构

MPLAB C18 支持联合内的匿名结构。匿名结构有以下形式：

struct { *member-list* };

匿名结构定义未命名的对象。匿名结构的成员名不能与定义此匿名结构的作用域内其它名称相同。在此作用域内，可以直接使用成员而无需使用通常的成员访问语法。

例如：

```
union foo
{
  struct
  {
    int a;
    int b;
  };
  char c;
} bar;
char c;

...

bar.a = c; /* 'a' is a member of the anonymous structure
              located inside 'bar' */
```

定义了对象或指针的结构不是匿名结构。例如：

```
union foo
{
  struct
  {
    int a;
    int b;
  } f, *ptr;
  char c;
} bar;
char c:

...

bar.a = c;      /* error */
bar.ptr->a = c; /* ok */
```

对 `bar.a` 的赋值是非法的，因为此成员名与任何特定的对象都没有关联。

2.8.2 行内汇编

MPLAB C18 提供了一个内部汇编器，它使用和 MPASM 汇编器相似的语法。汇编代码块必须以 _asm 开头，以 _endasm 结尾。其语法如下：

```
[label:] [<instruction> [arg1[, arg2[, arg3]]]]
```

内部汇编器与 MPASM 汇编器的差别如下：

- 不支持伪指令
- 注释必须使用 C 或 C++ 符号
- 表格读 / 写必须使用全文本助记符，即：
  - TBLRD
  - TBLRDPOSTDEC
  - TBLRDPOSTINC
  - TBLRDPREINC
  - TBLWT
  - TBLWTPOSTDEC
  - TBLWTPOSTINC
  - TBLWTPREINC
- 没有默认的指令操作数 — 必须完整地指定所有操作数
- 默认的数制是十进制。
- 使用 C 的数制符号表示常数，而不是 MPASM 汇编器的符号。例如，一个十六进制数应表示为 0x1234，而不是 H’1234’。
- 标号必须包含冒号
- 不支持变址寻址语法 （即 []）— 必须指定立即数和存取位 （例如指定为 CLRF 2,0，而不是 CLRF [2]）

例如：

```
_asm
  /* User assembly code */
  MOVLW 10       // Move decimal 10 to count
  MOVWF count, 0

  /* Loop until count is 0 */
  start:
    DECFSZ count, 1, 0
    GOTO done
    BRA start
  done:
_endasm
```

一般情况下，建议尽量少使用行内汇编。编译器不会优化任何包含行内汇编的函数。如果要编写大段的汇编代码，应使用 MPASM 汇编器，并用 MPLINK 链接器把汇编模块链接到 C 模块。

2.9 PRAGMA 伪指令

2.9.1 #pragma *sectiontype*

段声明 pragma 伪指令将当前段更改为 MPLAB C18 分配相关类型的信息的段。

段是位于特定存储器地址的应用程序的一部分。段可以包含代码或数据，可以位于程序存储器或数据存储器中。对于每种存储器，都有两种段类型。

- 程序存储器
  - **code** – 包含可执行指令
  - **romdata** – 包含变量和常量
- 数据存储器
  - **udata**– 包含静态分配的未初始化用户变量
  - **idata**– 包含静态分配的已初始化用户变量

段分为绝对的、已分配的或未分配的。绝对段是指通过段声明 pragma 伪指令的 `=`*address* 明确赋予了地址的段。已分配段是指已通过链接器描述文件中的 `SECTION` 伪指令分配到某个特定段的段。未分配段既不属于绝对段，也不属于已分配段。

2.9.1.1 语法

段伪指令：

```
  # pragma udata [ 属性列表 ] [section-name [=address]]
| # pragma idata [ 属性列表 ] [section-name [=address]]
| # pragma romdata [overlay] [section-name [=address]]
| # pragma code [overlay] [section-name [=address]]
```

属性列表：

属性
| *属性列表 属性*

属性：

```
  access
| overlay
```

section-name： C 标识符

address：整型常量

2.9.1.2 段内容

`code` 段包含可执行的内容，位于程序存储器中。 `romdata` 段包含分配到程序存储器的数据 （一般是用 `rom` 限定符定义的变量）。若需要有关 `romdata` 用法 （例如存储器映射外设）的更多信息，请参阅 *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) 中的 MPLINK linker 部分。 `udata` 段包含静态分配到数据存储器的未初始化全局数据。 `idata` 段包含静态分配到数据存储器的已初始化全局数据。

表 2-6 列出了下例中的每个对象位于哪个段中：

```
rom int ri;
rom char rc = 'A';

int ui;
char uc;

int ii = 0;
char ic = 'A';

void foobar (void)
{
  static rom int foobar_ri;
  static rom char foobar_rc = 'Z';
  ...
}
void foo (void)
{
  static int foo_ui;
  static char foo_uc;
  ...
}

void bar (void)
{
  static int bar_ii = 5;
  static char bar_ic = 'Z';
  ...
}
```

表 **2-6**： 对象的段位置

对象	位置
ri	romdata
rc	romdata
foobar_ri	romdata
foobar_rc	romdata
ui	udata
uc	udata
foo_ui	udata
foo_uc	udata
ii	idata
ic	idata
bar_ii	idata
bar_ic	idata
foo	code
bar	code
foobar	code

2.9.1.3 默认段

在 MPLAB C18 中，每种段类型都存在默认段 （见表 2-7 ）。

表 **2-7**： 默认段名

段类型	默认名称
code	.code_*filename*
romdata	.romdata_*filename*
udata	.udata_*filename*
idata	.idata_*filename*

注： *filename* 是所生成目标文件的名称。例如，“mcc18 foo.c -fo=foo.o”将生成一个目标文件，默认代码段名为“.code_foo.o”。

指定一个先前声明过的段名将使 MPLAB C18 重新把相关类型的数据分配到指定的段。段属性必须与先前的声明一致，否则会产生错误 （请参见**附录 D.1“错误”**。）

不带段名的段 pragma 伪指令把相关类型的数据分配到当前模块的默认段。例如：

```
/*
 * The following statement changes the current code
 * section to the absolute section high_vector
 */
#pragma code high_vector=0x08
...

/*
 * The following statement returns to the default code
 * section
 */
#pragma code
...
```

MPLAB C18 编译器开始编译一个源文件时，初始化的数据和未初始化的数据都有默认数据段。这些默认段位于存取 RAM 或非存取 RAM，这取决于是否使用了`-Oa+` 选项调用编译器。仅当编译器工作在非扩展模式 （参见 **1.2.5 节 “选择模式”**）时才使用`-Oa+` 命令行选项。当在源代码中遇到一个 `#pragma idata [access]` *name* 伪指令时，当前未初始化数据段的名称就成为 *name*，它位于存取 RAM 或非存取 RAM，这取决于是否指定了可选的 *access* 属性。对于当前已初始化数据段，当遇到一个 `#pragma idata [`*access*`]` *name* 伪指令时，与上述情况相同。

当对象定义中有显式的初始化时，对象被放入当前已初始化数据段中。当对象定义中没有显式的初始化时，对象被放入当前未初始化数据段中。例如，在以下的代码片段中， `i` 将放入当前已初始化数据段中，而 `u` 将被放在当前未初始化数据段中。

```
int i = 5;
int u;

void main(void)
{
   ...
}
```

如果对象的定义有显式的 `far` 限定符 （参见 **2.4 节 “存储限定符”**），对象存放在非存取存储区。类似地，显式的 `near` 限定符 （参见 **2.4 节 “存储限定符”**）告知编译器对象位于存取存储区。如果对象的定义既没有 `near` 限定符也没有 `far` 限定符，则编译器将查看是否在命令行中指定了`-Oa+` 选项。

2.9.1.4 段属性

#pragma *sectiontype* 伪指令可以选择包含两种段属性 — access 或 overlay。

2.9.1.4.1 access

access 属性告知编译器把指定的段放入数据存储器的存取区中 （参见器件数据手册或 《*PICmicro® 18C 单片机系列参考手册*》（DS39500A_CN），以获得更多关于存取数据存储区的信息）。

带有 access 属性的数据段将放入在链接器描述文件中定义为 ACCESSBANK 的存储区。这些存储区可以通过指令的存取位来访问，也就是说，不需要选择存储区 （参见器件数据手册）。 access 段中的变量必须用 near 关键字定义。例如：

```
#pragma udata access my_access
/* all accesses to these will be unbanked */
near unsigned char av1, av2;
```

2.9.1.4.2 overlay

overlay 属性允许其它段与此段位于相同的物理地址。通过把变量放入相同的存储单元可节约存储空间 （只要两个变量不会被同时使用）。overlay 属性可与 access 属性一起使用。

若要使两个段共享相同的存储空间，必须满足如下四个条件：

1. 两个段必须位于不同的源文件中。
2. 两个段必须有相同的名称。
3. 如其中一个段已指定 access 属性，那么也必须对另一个段指定 access 属性。
4. 如果其中一个段已指定绝对地址，那么也必须对另一个段指定相同的绝对地址。

具有 overlay 属性的代码段可以位于与其它 overlay 代码段重叠的地址中。例如：

file1.c:

```
#pragma code overlay my_overlay_scn=0x1000
void f (void)
{
 ...
}
```

file2.c:

```
#pragma code overlay my_overlay_scn=0x1000
void g (void)
{
 ...
   }
```

具有 overlay 属性的数据段可以位于与其它 overlay 数据段重叠的地址中。这一特征是很有用的，可允许确定不会同时使用的多个变量使用同一个数据存储区。例如：

file1.c:

```
#pragma udata overlay my_overlay_data=0x1fc
/* 2 bytes will be located at 0x1fc and 0x1fe */
int int_var1, int_var2;
```

file2.c:

```
#pragma udata overlay my_overlay_data=0x1fc
/* 4 bytes will be located at 0x1fc */
long long_var;
```

更多关于处理重叠段的信息，可参阅 *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014)。

2.9.1.5 定位代码

在 #pragma code 伪指令后生成的所有代码将被分配到指定的代码段，直到遇到下一个 #pragma code 伪指令。绝对代码段允许将代码分配到一个特定的地址。例如：

```
#pragma code my_code=0x2000
```

将把代码段 my_code 分配到程序存储器地址 0x2000。

链接器会强制将代码段放入程序存储区；然而，代码段也可以位于指定的存储区。可以用链接器描述文件中的 SECTION 伪指令把一个段分配到特定的存储区。下面链接器描述文件中的伪指令把代码段 my_code1 分配到存储区 page1：

```
SECTION NAME=my_code1 ROM=page1
```

2.9.1.6 定位数据

对于 MPLAB C18 编译器，数据可以放入数据存储器或者程序存储器。如果没有用户提供的附加代码，片内程序存储器中的数据只能读不能写。如果没有用户提供的附加代码，片外程序存储器中的数据一般是只能读或者只能写。

例如，下面的语句为静态分配的未初始化数据 （udata）声明了一个位于绝对地址 0x120 的段：

```
#pragma udata my_new_data_section=0x120
```

rom 关键字告知编译器应该将变量放入程序存储器。编译器会把这个变量分配到当前的 romdata 型段。例如：

```
#pragma romdata const_table
const rom char my_const_array[10] = {0, 1, 2, 3, 4, 5,
                                     6, 7, 8, 9};

/* Resume allocation of romdata into the default section */
#pragma romdata
```

链接器强制将 romdata 段放入程序存储区，将 udata 和 idata 段放入数据存储区；然而，数据段也可以位于指定的存储区。可以使用链接器描述文件中的 SECTION 伪指令把一个段分配到一个特定的存储区。下面的语句将把 udata 段 my_data 分配到存储区 gpr1：

```
SECTION NAME=my_data RAM=gpr1
```

2.9.2 #pragma interruptlow *fname* / #pragma interrupt *fname*

interrupt pragma 伪指令将函数声明为高优先级的中断服务程序； interruptlow pragma 伪指令将函数声明为低优先级的中断服务程序。

中断将暂停执行正在运行的应用程序，保存当前的现场信息并把控制权转交给中断服务程序，以便对事件进行处理。执行完中断服务程序后，恢复先前的现场信息，继续正常执行应用程序。对于中断来说，可保存和恢复的最小现场是 WREG、 BSR 和 STATUS。高优先级中断使用影子寄存器保存和恢复最小现场，而低优先级中断则使用软件堆栈保存和恢复最小现场。因此，高优先级中断可通过一个快速 “中断返回”来结束，而低优先级中断则通过一个一般 “中断返回”来结束。通过软件堆栈保存现场的每个字节需要两条 MOVFF 指令， WREG 例外，它需要一条 MOVWF 指令和一条 MOVF 指令；因此，要保存最小现场，一个低优先级中断要比一个高优先级中断多开销 10 个字。

中断服务程序使用一个临时数据段，这个段和一般 C 函数所使用的段是不同的。中断服务程序中，在表达式求值过程中所需要的所有临时数据都被分配到此段，而且此段不与其它函数 （包括其它中断函数）的临时地址重叠。中断 pragma 伪指令允许给中断临时数据段命名。如果没有给这个段命名，将会在名为 *fname*_tmp 的 udata 段中生成编译器临时变量。例如：

```
void foo(void);
...
#pragma interrupt foo
void foo(void)
{
  /* perform interrupt function here */
}
```

中断服务程序 foo 的编译器临时变量将被放在 udata 段 foo_tmp 中。

2.9.2.1 语法

中断伪指令：

```
# pragma interrupt function-name
  [tmp-section-name][save=save-list]
| # pragma interruptlow function-name
  [tmp-section-name][save=save-list]
```

save-list:

```
  save-specifier
| save-list, save-specifier
```

save-specifier:

```
  symbol-name
| section("section-name")
```

`function-name`：C 标识符 — 命名作为中断服务程序的 C 函数。

`tmp-section-name`：C 标识符 — 命名分配中断服务程序临时数据的段。

`symbol-name`：C 标识符 — 命名中断处理后恢复的变量。

`section-name`：C 标识符，与一般标识符的不同之处是第一个字符可以是一个点（.）— 命名中断处理后恢复的段。

2.9.2.2 中断服务程序

MPLAB C18 中断服务程序与任何其它 C 函数一样，也可以有局部变量并能访问全局变量；然而，中断服务程序必须定义为没有参数和返回值，这是因为响应硬件中断的中断服务程序是异步调用的。既可以被中断服务程序访问也可以被其它函数访问的全局变量应定义为 `volatile`。

中断服务程序只能通过硬件中断调用，而不能从其它 C 函数中调用。中断服务程序使用中断返回指令 （`RETFIE`）退出中断函数，而不是使用一般的 `RETURN` 指令。不恢复现场使用快速 `RETFIE` 指令退出中断服务程序，会破坏 `WREG`、 `BSR` 和 `STATUS` 寄存器的值。

2.9.2.3 中断向量

MPLAB C18 不自动把中断服务程序放在中断向量处。通常将 GOTO 指令放在中断向量处，从而把控制权转交给相应的中断服务程序。例如：

```
#include <p18cxxx.h>

void low_isr(void);
void high_isr(void);

/*
 * For PIC18 devices the low interrupt vector is found at
 * 00000018h. The following code will branch to the
 * low_interrupt_service_routine function to handle
 * interrupts that occur at the low vector.
 */
#pragma code low_vector=0x18
void interrupt_at_low_vector(void)
{
  _asm GOTO low_isr _endasm
}
#pragma code /* return to the default code section */

#pragma interruptlow low_isr
void low_isr (void)
{
  /* ... */
}

/*
 * For PIC18 devices the high interrupt vector is found at
 * 00000008h. The following code will branch to the
 * high_interrupt_service_routine function to handle
 * interrupts that occur at the high vector.
 */
#pragma code high_vector=0x08
void interrupt_at_high_vector(void)
{
  _asm GOTO high_isr _endasm
}
#pragma code /* return to the default code section */

#pragma interrupt high_isr
void high_isr (void)
{
  /* ... */
}
```

如果需要完整的例子，可参阅**第 5 章 “示例应用程序”**。

2.9.2.4 中断现场保护

默认情况下， MPLAB C18 保存基本的现场 （参见 **3.4 节 “编译器管理的资源”**）， save= 子句允许函数保存和恢复其它任意符号。

要保存一个名为 myint 的用户定义全局变量，使用如下 pragma 伪指令：

```
#pragma interrupt high_interrupt_service_routine save=myint
```

除了变量，也可以在 save= 子句中指定整个数据段。比如，要保存一个名为 mydata 的用户定义段，使用如下 pragma 伪指令：

```
#pragma interrupt high_interrupt_service_routine
save=section("mydata")
```

如果一个中断服务程序调用另一个函数，要保存这个普通函数的临时数据段 （名为 .tmpdata），应该在中断 pragma 伪指令中使用 save=section(".tmpdata") 限定符。例如：

```
#pragma interrupt high_interrupt_service_routine
save=section(".tmpdata")
```

如果中断服务程序改变了除基本现场外的任何数据寄存器，那么应在 save= 子句中指定它们。应该检查所生成的代码，以确定哪些数据寄存器被使用过并需要保存。

> **注：** 如果中断服务程序调用一个函数，此函数返回值的长度小于或等于 32 位，则应该在中断 pragma 伪指令的 save= 列表中指定与此返回值相对应的位置 （参见 **3.2.1 节 “返回值”**）。

如果中断服务程序调用了一个返回 16 位数据的函数，应该在中断 pragma 伪指令中使用 save=PROD 限定符保存 PROD 数据寄存器 。例如：

```
#pragma interruptlow low_interrupt_service_routine save=PROD
```

如果中断服务程序使用数学库函数或者调用一个返回 24 位或 32 位数据的函数，应该在中断 pragma 伪指令中使用 save=section("MATH_DATA") 限定符保存数学数据段 （名为 MATH_DATA）。例如：

```
#pragma interrupt high_interrupt_service_routine save=section("MATH_DATA")
```

上述所有例子都是保存单个值，也可以使用 save= 限定符保存多个变量和段。如果一个中断服务程序使用了 PROD 数据寄存器、 .tmpdata 段、 myint 变量和 mydata 段，则应该在中断 pragma 伪指令中使用 save=PROD, section (".tmpdata"), myint, section ("mydata") 限定符来保存它们 。例如：

```
#pragma interrupt isr save=PROD, section(".tmpdata"), myint, section("mydata")
```

2.9.2.5　中断响应时间

从中断发生到执行中断服务程序第一条指令之间的时间就是中断响应时间。中断响应时间受到下面三个因素的影响：

1. **处理器中断处理时间**：处理器识别中断并跳转到中断向量的首地址所用的时间。要确定这个值的大小，可以参考相应单片机的数据手册，以获得该型号单片机及所使用中断源方面的信息。
2. **中断向量执行**：执行中断向量处、跳转到中断服务程序的代码所用的时间。
3. **ISR 预处理的时间**：MPLAB C18 保存编译器管理的资源和 `save=` 列表中的数据所用的时间。

2.9.2.6　中断嵌套

低优先级中断可以嵌套，这是因为所使用寄存器的值保存在软件堆栈中。任一时刻只能有一个高优先级中断服务程序的实例处于运行状态，因为高优先级中断服务程序使用单级深度硬件影子寄存器。

如果需要嵌套低优先级中断，可以在中断服务程序的开头部分加上一条置位 `GIEL` 位的语句。详情请参阅单片机的数据手册。

2.9.3　#pragma varlocate *bank variable-name* #pragma varlocate "*section-name*" *variable-name*

`varlocate` 告知编译器在链接时变量的位置，这使编译器能更高效地执行存储区切换。

`varlocate` 说明不是由编译器或链接器强制执行的。应在链接器描述文件中把包含变量的段明确地分配到正确的存储区，或者在定义这些变量的模块中把包含变量的段分配到绝对段。

2.9.3.1　语法

变量定位伪指令 :

```
  # pragma varlocate bank variable-name[, variable-name...]
| # pragma varlocate "section-name" variable-name[,
  variable-name...]
```

bank : 整型常量

variable-name：C 标识符

section-name：C 标识符

2.9.3.2 使用 # pragma varlocate *bank variable-name* 的例子

在一个文件中，把 c1 和 c2 显式地分配到存储区 1。

```
#pragma udata bank1=0x100
signed char c1;
signed char c2;
```

在第二个文件中，编译器被告知 c1 和 c2 都位于存储区 1。

```
#pragma varlocate 1 c1
extern signed char c1;

#pragma varlocate 1 c2
extern signed char c2;

void main (void)
{
  c1 += 5;
  /* No MOVLB instruction needs to be generated here. */
  c2 += 5;
}
```

当在第二个文件中使用 c1 和 c2 时，编译器知道这两个变量都在相同的存储区内，当在 c1 后紧接着使用 c2 时就不需要再生成另外一条 MOVLB 指令了。

2.9.3.3 使用 # pragma varlocate "*section-name*" *variable-name* 的例子

在一个文件中，c3 和 c4 都创建在 udata 段 my_section 中。

```
#pragma udata my_section
signed char c3;
signed char c4;
#pragma udata
```

在第二个文件中，编译程序被告知 c3 和 c4 都位于 udata 段 my_section 中。

```
#pragma varlocate "my_section" c3, c4
extern signed char c3;
extern signed char c4;

void main (void)
{
  c3 += 5;
  /* No MOVLB instruction needs to be generated here. */
  c4 += 5;
}
```

当在第二个文件中用到 c3 和 c4 时，编译器知道这两个变量都在相同的段中，当在 c3 后紧接着使用 c4 时就不需要再生成另外一条 MOVLB 指令了。

2.10 针对处理器的头文件

针对处理器的头文件是包含特殊功能寄存器外部声明的 C 文件，特殊功能寄存器在寄存器定义文件中定义 （参见 **2.11 节 "针对处理器的寄存器定义文件"**）。例如，在 PIC18C452 的针对处理器头文件中， PORTA 声明为：

```
extern volatile near unsigned char PORTA;
```

和：

```
extern volatile near union  {
  struct {
    unsigned RA0:1;
    unsigned RA1:1;
    unsigned RA2:1;
    unsigned RA3:1;
    unsigned RA4:1;
    unsigned RA5:1;
    unsigned RA6:1;
  } ;
  struct {
    unsigned AN0:1;
    unsigned AN1:1;
    unsigned AN2:1;
    unsigned AN3:1;
    unsigned T0CKI:1;
    unsigned SS:1;
    unsigned OSC2:1;
  } ;
  struct {
    unsigned :2;
    unsigned VREFM:1;
    unsigned VREFP:1;
    unsigned :1;
    unsigned AN4:1;
    unsigned CLKOUT:1;
  } ;
  struct {
    unsigned :5;
    unsigned LVDIN:1;
  } ;
} PORTAbits ;
```

第一个声明指定 `PORTA` 是一个字节 （`unsigned char`）。由于变量是在寄存器定义文件中定义的，因此需要 `extern` 修饰符。 `volatile` 修饰符告知编译器不能假定 PORTA 能保留赋给它的值。 `near` 修饰符指定了端口位于存取 RAM 中。

第二个声明指定 `PORTAbits` 是可位寻址的匿名结构的联合 （参见 **2.8.1 节 "匿名结构"**）。特殊功能寄存器中的每一位可能有不只一种功能 （因此会有不只一个名称），因此联合中对于同一个寄存器有多个结构定义。所有结构定义中的各位分别针对寄存器中相同的位。如果一个位只有一个功能，那么在其它结构定义中，这一位只是被填充。例如，在第三和第四个结构中， PORTA 的第 1 位 和第 2 位只是被填充，因为它们只有两个名称；而第 6 位有四个名称，在每个结构中都指定了第 6 位。

可用以下任一语句使用特殊功能寄存器 PORTA：

```
PORTA = 0x34;      /* Assigns the value 0x34 to the port */
PORTAbits.AN0 = 1; /* Sets the AN0 pin high */
PORTAbits.RA0 = 1; /* Sets the RA0 pin high, same as above
                      statement */
```

除了寄存器声明外，针对处理器的头文件还定义了行内汇编宏。这些宏代表某些 PICmicro 单片机指令，应用程序可能需要从 C 代码中执行这些指令。尽管这些指令可以作为行内汇编指令引用，但为方便起见，提供了 C 宏 （见表 2-8）。

最后，针对处理器的头文件包含宏 _CONFIG_DECL 和一些允许指定器件配置的每个设置的 #define 语句。更多信息请参阅 **2.12 节 “配置字”**。

为了使用针对处理器的头文件，选择适合所使用器件的头文件 （例如，如果使用 PIC18C452，应该在应用源代码中使用 #include <p18c452.h>）。针对处理器的头文件位于 c:\mcc18\h 目录下，其中 c:\mcc18 是安装编译器的目录。另一种方法是，根据通过-p 命令行选项在命令行上选择的处理器，使用 #include <p18cxxx.h> 包含正确的针对处理器头文件。

表 2-8： 为 PICmicro MCU 指令提供的 C 宏

指令宏 [1]	作用
Nop()	执行一个空操作 （NOP）
ClrWdt()	清零看门狗定时器 （CLRWDT）
Sleep()	执行一条 SLEEP 指令
Reset()	执行器件复位 （RESET）
Rlcf(*var, dest, access*)[2,3]	*var* 带进位循环左移
Rlncf(*var, dest, access*)[2,3]	*var* 不带进位循环左移
Rrcf(*var, dest, access*)[2,3]	*var* 带进位循环右移
Rrncf(*var, dest, access*)[2,3]	*var* 不带进位循环右移
Swapf(*var, dest, access*)[2,3]	交换 *var* 的前半字节和后半字节

注 **1：** 在一个函数中使用这些宏，会影响 MPLAB C18 编译器对这个函数的优化。

2： *var* 必须是一个 8 位量 (即 char)，并且不在堆栈中。

3： 如果 *dest* 为 0，结果保存在 WREG 中；如果 *dest* 为 1，结果保存在 *var* 中。如果 *access* 为 0，则选中存取存储区，与 BSR 的值无关。如果 access 为 1，那么会按照 BSR 的值选择存储区。

2.11 针对处理器的寄存器定义文件

针对处理器的寄存器定义文件是一个汇编文件，包含特定器件上所有特殊功能寄存器的定义。编译时，针对处理器的寄存器定义文件将被编译成需要链接到应用程序的目标文件 （例如， p18c452.asm 编译为 p18c452.o）。此目标文件包含在 p18*xxxx*.lib 中 （例如， p18c452.o 包含在 p18c452.lib 中）。

针对处理器的寄存器定义文件的源代码在 c:\mcc18\src\traditional\proc 和 c:\mcc18\src\extended\proc 目录下。编译后的目标代码在 c:\mcc18\lib 目录下，而 c:\mcc18 就是安装编译器的目录。

例如，在 PIC18C452 针对处理器的寄存器定义文件中， PORTA 定义为：

```
SFR_UNBANKED0 UDATA_ACS H'f80'
PORTA
PORTAbits RES 1 ; 0xf80
```

第一行指定 PORTA 所在的数据寄存器存储区和这个存储区的起始地址。 PORTA 有两个标号， PORTAbits 和 PORTA，都指向同一个地址 （本例中是 0xf80）。

2.12 配置字

每一器件的默认链接器描述文件都含有一个名为 CONFIG 的段。例如，p18c452.lkr 脚本包含以下语句：

```
CODEPAGE NAME=config START=0x300000 END=0x300007 PROTECTED
...
SECTION NAME=CONFIG ROM=config
```

#pragma romdata CONFIG 伪指令用于把当前的 romdata 段设置成名为 CONFIG 的段。可以使用 _CONFIG_DECL 宏和针对处理器头文件中的 # define 语句指定器件的配置。下面的例子指定了表 2-9 的配置。

```
#include <p18c452.h>
#pragma romdata CONFIG
_CONFIG_DECL (_CP_ON_1L,
              _OSCS_ON_1H & _OSC_LP_1H,
              _PWRT_ON_2L & _BOR_OFF_2L & _BORV_42_2L,
              _WDT_OFF_2H & _WDTPS_1_2H,
              _CCP2MUX_OFF_3H,
              _CONFIG4L_DEFAULT);
#pragma romdata
void main (void)
{
  ...
}
```

表 **2-9**： 配置实例

设置	指定的配置
`_CP_ON_1L`	所有程序存储器都是代码保护的
`_OSCS_ON_1H`	使能振荡器系统时钟切换选项 （允许切换振荡器）
`_OSC_LP_1H`	LP 振荡器
`_PWRT_ON_2L`	使能上电延时定时器
`_BOR_OFF_2L`	禁止欠压复位
`_BORV_42_2L`	V_{BOR} 设置为 4.2V
`_WDT_OFF_2H`	禁止看门狗定时器
`_WDTPS_1_2H`	看门狗定时器后分频比为 1:1
`_CCP2MUX_OFF_3H`	RB3 引脚复用为 CCP2 输入 / 输出
`_CONFIG4L_DEFAULT`	此配置字节的默认设置 — 堆栈满 / 下溢将导致复位



第 3 章 运行时模型

本章讲述 MPLAB C18 编译器的运行时模型及运行所遵循的各项前提，包括 MPLAB C18 编译器如何使用 PIC18 PICmicro 单片机的资源等信息。

3.1 存储模型

MPLAB C18 同时为小存储模型和大存储模型（见表 3-1 ）提供全部库支持。使用-ms 命令行选项选择小存储模型，使用-ml 命令行选项选择大存储模型。如果不使用这两个选项，默认使用小存储模型。

表 **3-1**： 存储模型概括

存储模型	命令行选项	默认的 **ROM** 范围限定符	指向程序空间 指针的长度
小存储模型	-ms	near	16 位
大存储模型	-ml	far	24 位

小存储模型和大存储模型之间的区别在于指向程序存储器的指针的长度不同。在小存储模型中，指向程序存储器的函数指针和数据指针都是 16 位的。因此在小存储模型中，指针被限定为只能在程序存储器的前 64k 范围内寻址。大存储模型使用 24 位长度的指针。若应用程序要使用大于 64k 的程序存储器地址，必须使用大存储模型。

定义一个指向程序空间的指针时，可用 near 或 far 限定符逐项改变存储模型的设置。指向 near 存储器的指针长度为 16 位，和在小存储模型中一样；而指向 far 存储器的指针长度为 24 位，和在大存储模型中一样。

下面的例子创建一个指向程序存储器的指针，即使在使用小存储模型时，它都能寻址达到和超过 64k 的程序存储空间[1]：

```
far rom *pgm_ptr;
```

下面的例子创建了一个函数指针，即使在使用小存储模型时，它都能寻址达到和超过 64k 的程序存储空间[2]：

```
far rom void (*fp) (void);
```

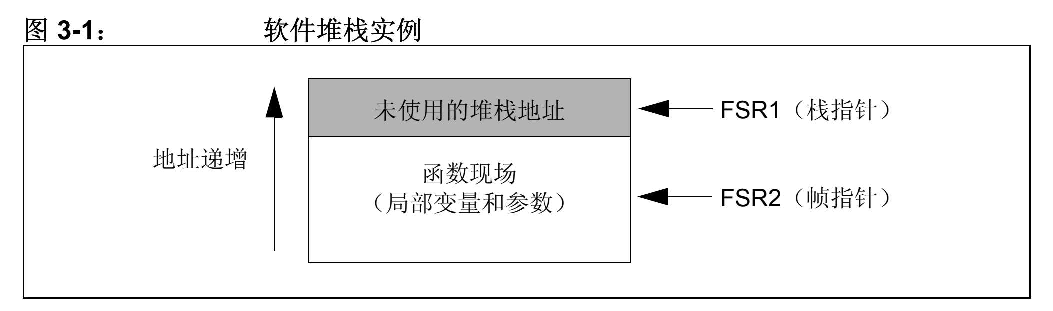
如果项目中的所有文件并非都使用相同的存储模型，那么所有指向程序存储器的全局指针都应该用 near 或 far 限定符显式地定义，这样才能在所有模块中正确地访问这些指针。小存储模型和大存储模型都可以使用 MPLAB C18 的预编译库。

1. 在小存储模型程序中使用 far 数据指针后，TBLPTRU 字节必须由用户清除，MPLAB C18 不会清除这个字节。
2. 在小存储模型程序中使用far函数指针后，PCLATU 字节必须由用户清除，MPLAB C18不会清除这个字节。

3.2 关于调用的约定

MPLAB C18 的软件堆栈是向上生长的堆栈数据结构，编译器把函数参数和 `auto` 存储类别的局部变量放入软件堆栈中。软件堆栈与 PICmicro 单片机用于保存函数调用返回地址的硬件堆栈不同。图 3-1 给出了一个软件堆栈的实例。

图 3-1： **软件堆栈实例**



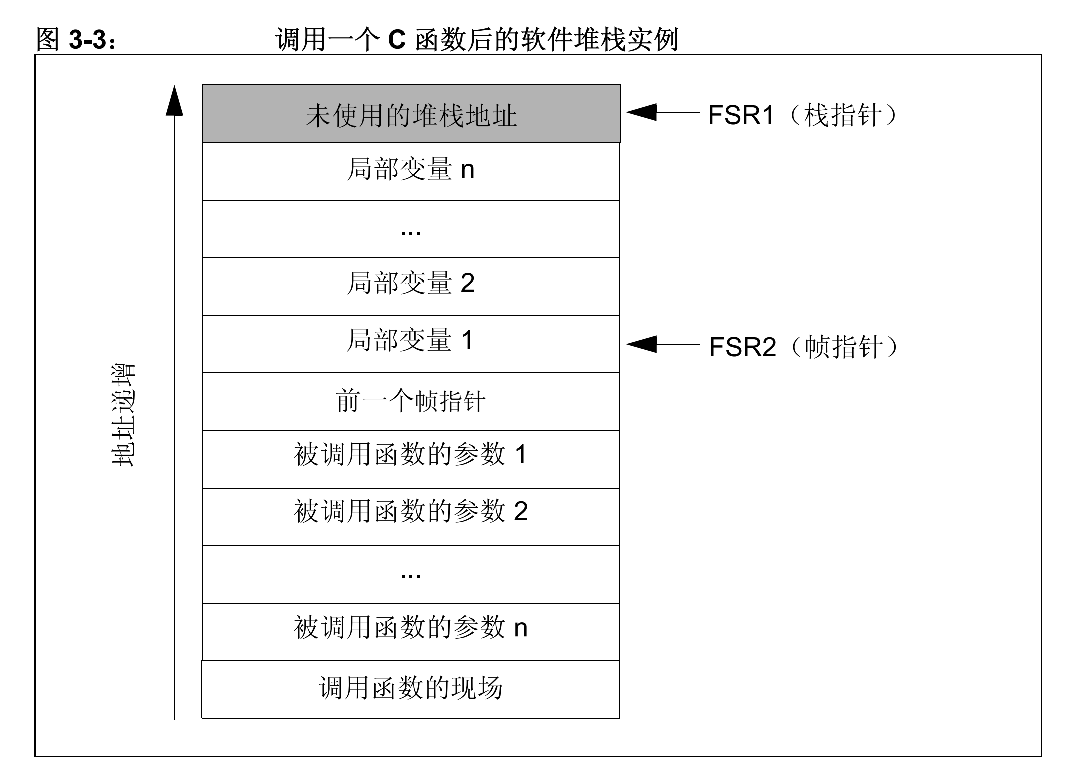
栈指针 （FSR1）始终指向下一个可用的堆栈地址。 MPLAB C18 使用 FSR2 作为帧指针，这样可以快速访问局部变量和参数。函数被调用时，其基于堆栈的参数以自右向左的顺序压入堆栈，然后再调用这个函数。进入函数时，最左端的函数参数位于软件堆栈的顶端。图 3-2 显示了函数调用前的软件堆栈。

图 3-2： **函数调用前的软件堆栈示例**

地址递增	堆栈	指针
↑	未使用的堆栈地址	← FSR1（栈指针）
	函数参数 1	
	函数参数 2	
	...	
	函数参数 n	
	函数现场	← FSR2（帧指针）

帧指针指向堆栈中把基于堆栈的参数和基于堆栈的局部变量分隔开的地址。基于堆栈的参数位于帧指针的下方，而基于堆栈的局部变量位于帧指针的上方。刚进入 C 函数时，被调用函数把 FSR2 的值压入堆栈，并把 FSR1 的值复制到 FSR2，从而保存了调用函数的现场并初始化了当前函数的帧指针。然后函数基于堆栈的局部变量的总长度被加到栈指针，并为这些变量分配堆栈空间。基于堆栈的局部变量和基于堆栈的参数则根据其相对于帧指针的偏移量来引用。图 3-3 显示调用一个 C 函数后的软件堆栈。

图 **3-3**： 调用一个 **C** 函数后的软件堆栈实例



3.2.1 返回值

返回值的位置取决于返回值的长度。表 3-2 详细列出了返回值的位置与其长度的关系。

表 **3-2**： 返回值

返回值的长度	返回值的位置
8 位	WREG
16 位	PRODH:PRODL
24 位	[非扩展模式] (AARGB2+2):(AARGB2+1):AARGB2 [扩展模式] __RETVAL2:__RETVAL1:__RETVAL0
32 位	[非扩展模式] (AARGB3+3):(AARGB3+2):(AARGB3+1):AARGB3 [扩展模式] __RETVAL3:__RETVAL2:__RETVAL1:__RETVAL0
> 32 位	位于堆栈中，而且 FSR0 指向返回值

3.2.2 管理软件堆栈

通过链接器描述文件中的 STACK 伪指令确定堆栈的长度和地址。 STACK 伪指令有两个参数：SIZE 和 RAM，分别用来设置堆栈的长度和地址。例如，分配一个 128 字节的堆栈并把此堆栈放入存储区 gpr3 中：

```
STACK SIZE=0x80 RAM=gpr3
```

MPLAB C18 支持大于 256 字节的堆栈。默认链接器描述文件为每个存储区都分配了一个存储区，因为堆栈区不能跨越存储区的边界，若要分配大于 256 字节的堆栈就需要组合两个或更多个存储区。例如， PIC18C452 的默认链接器描述文件包含以下定义：

```
DATABANK NAME=gpr4 START=0x400 END=0x4ff
DATABANK NAME=gpr5 START=0x500 END=0x5ff
...
STACK SIZE=0x100 RAM=gpr5
```

要把一个 512 字节的堆栈分配到存储区 4 和 5 中，应将上述定义更改为：

```
DATABANK NAME=stackregion START=0x400 END=0x5ff PROTECTED
STACK SIZE=0x200 RAM=stackregion
```

如果要使用大于 256 字节的堆栈，必须向编译器提供-ls 选项。使用大型堆栈时会对性能造成轻微的不利影响：当执行 push/pop （压栈／出栈）操作时，帧指针的两个字节 （FSR2L 和 FSR2H）都必须递增／递减，而不只是递增／递减 FSR2L。

根据程序的复杂程度不同，应用程序需要的软件堆栈的大小也不同。当嵌套调用函数时，调用函数的所有 auto 参数和变量仍然保存在堆栈中。因此，堆栈必须足够大，以满足调用树中所有函数的要求。

MPLAB C18 支持把参数和局部变量分配到软件堆栈中或直接分配到全局存储区中。static 关键字将局部变量或函数参数放入全局存储区，而不是放入软件堆栈。[1] 一般来说，存取基于堆栈的局部变量和函数参数会比存取 static 局部变量和函数参数需要更多的代码 （参见 **2.3.2 节 “static 型函数参数”**）。使用基于堆栈的变量的函数更为灵活，因为它们是可重入的和／或可递归的。

3.2.3 C 语言与汇编语言的混合编程

3.2.3.1 在汇编程序中调用 C 函数

在汇编程序中调用 C 函数时：

- C 函数本质上是全局的，除非定义为 static。
- 必须在汇编文件中用 extern 声明 C 函数。
- 调用函数必须使用 CALL 或 RCALL 指令。

1. static 参数仅当编译器工作在非扩展模式 （参见 **1.2.5 节 “选择模式”**）时有效。

3.2.3.1.1 **auto** 型参数

auto 型参数以从右到左的顺序压入软件堆栈。对于多字节数据来说，首先将低字节压入软件堆栈。

例 3-1：

下面给出一个 C 函数的原型：

```
char add (auto char x, auto char y);
```

要用值 x = 0x61 和 y = 0x65 调用函数 add，必须先将 y 的值压入软件堆栈，然后将 x 的值压入软件堆栈。由于返回值长度为 8 位，所以返回到 WREG 中 （参见表 3-2），代码如下：

```
                EXTERN add  ; defined in C module
...
MOVLW 0x65
MOVWF POSTINC1 ; y = 0x65 pushed onto stack
MOVLW 0x61
MOVWF POSTINC1 ; x = 0x61 pushed onto stack
CALL  add
MOVWF result   ; result is returned in WREG
...
```

例 3-2：

下面给出一个 C 函数的原型：

```
int sub (auto int x, auto int y);
```

要用值 x = 0x7861 和 y = 0x1265 调用函数 sub，必须先将 y 的值压入软件堆栈，然后将 x 的值压入软件堆栈。由于返回值长度为 16 位，所以返回到 PRODH:PRODL 中（参见表 3-2），代码如下：

```
           EXTERN sub  ; defined in C module
...
MOVLW 0x65
MOVWF POSTINC1
MOVLW 0x12
MOVWF POSTINC1        ; y = 0x1265 pushed onto stack
MOVLW 0x61
MOVWF POSTINC1
MOVLW 0x78
MOVWF POSTINC1        ; x = 0x7861 pushed onto stack
CALL  sub
MOVFF PRODL, result
MOVFF PRODH, result+1 ; result is returned in PRODH:PRODL
...
```

3.2.3.1.2 static 参数

static 参数是全局分配的，可以直接访问。static 参数仅当编译器工作在非扩展模式（参见 **1.2.5 节 “选择模式”**）时有效。static 参数的命名规则是 *__function_name:n*，其中，*function_name* 表示函数名，而 *n* 为参数位置，从 0 开始计数。例如，下面给出一个 C 函数的原型：

```
char add (static char x, static char y);
```

可通过 __add:1 访问 y 的值，通过 __add:0 访问 x 的值。

注：	因为 ‘：’ 不是 MPASM 汇编器标号中的有效字符，所以不支持在汇编函数中访问 static 参数。

3.2.3.2 在 C 程序中调用汇编函数

在 C 程序中调用汇编函数时：

- 必须在汇编模块中将函数定义为 global。
- 必须在 C 模块中将函数声明为 extern。
- 函数必须保持 MPLAB C18 编译器的运行时模型（例如，必须把返回值返回到表 3-2 中指定的位置）。
- 在 C 程序中使用标准 C 函数符号调用汇编函数。

例 3-3：

下面是用汇编语言编写的函数：

```
       UDATA_ACS
delay_temp     RES     1

               CODE
asm_delay
          SETF   delay_temp
not_done
          DECF   delay_temp
          BNZ    not_done
done
          RETURN

                GLOBAL asm_delay  ; export so linker can see it
                END
```

要从 C 源文件中调用汇编函数 asm_delay，必须在 C 源文件中将此汇编函数声明为外部函数，并使用标准 C 函数符号调用它：

```
/* asm_delay is found in an assembly file */
extern void asm_delay (void);

void main (void)
{
  asm_delay ();
}
```


例 **3-4**：

以下是用汇编语言编写的函数：

```
               INCLUDE "p18c452.inc"

               CODE
asm_timed_delay
not_done
          ; Figure 3-2 is what the stack looks like upon
          ; entry to this function.
          ;
          ; ‘time’ is passed on the stack and must be >= 0
          MOVLW  0xff
          DECF   PLUSW1, 0x1, 0x0
          BNZ    not_done
done
          RETURN
               ; export so linker can see it
               GLOBAL asm_timed_delay
               END
```

要在 C 源文件中调用汇编函数 asm_timed_delay，必须在 C 源文件中将此汇编函数声明为外部函数，然后使用标准 C 函数符号调用它：

```
/* asm_timed_delay is found in an assembly file */
extern void asm_timed_delay (unsigned char);

void main (void)
{
  asm_timed_delay (0x80);
}
```

3.2.3.3 在汇编程序中使用 C 变量

要在汇编程序中使用 C 变量：

- 在 C 源文件中， C 变量必须有全局作用域。
- 必须在汇编文件中将 C 变量声明为 extern。

例 3-5：

以下是用 C 编写的程序：

```
unsigned int c_variable;

void main (void)
{
  ...
}
```

要在汇编程序中使用变量 c_variable，必须在汇编源文件中将其声明为外部变量：

```
        EXTERN c_variable  ; defined in C module
MYCODE CODE
asm_function
     MOVLW 0xff
     ; put 0xffff in the C declared variable
     MOVWF c_variable
     MOVWF c_variable+1
done
     RETURN

     ; export so linker can see it
     GLOBAL asm_function
     END
```

3.2.3.4 在 C 程序中使用汇编变量

在 C 程序中使用汇编变量时：

- 必须在汇编模块中将变量定义为 global。
- 必须在 C 模块中将变量声明为 extern。

例 3-6：

以下是用汇编语言编写的程序：

```
MYDATA UDATA
asm_variable RES   2 ; 2 byte variable

     ; export so linker can see it
     GLOBAL asm_variable
     END
```

要在 C 源文件中使用变量 asm_variable，必须在 C 源文件中将其声明为外部变量，可以象使用 C 变量一样使用变量 asm_variable：

```
extern unsigned int asm_variable;

void change_asm_variable (void)
{
  asm_variable = 0x1234;
}
```

3.3 启动代码

3.3.1 默认操作

MPLAB C18 从 reset 向量 （地址 0）处启动。 reset 向量跳转到一个函数，此函数初始化软件堆栈的栈指针 FSR1 和帧指针 FSR2 。用户还可选择调用一个初始化 idata 段 （数据存储器中的已初始化数据）的函数，然后循环调用 main() 函数。

启动代码是否会初始化 idata 段，取决于是哪个 （些）启动代码模块与应用程序链接。c018i.o 和 c018i_e.o 模块会进行初始化，而 c018.o 和 c018_e.o 模块不会。由 MPLAB C18 提供的默认链接器描述文件与 c018i.o 还是 c018i_e.o 模块链接，取决于使用的是非扩展模式还是扩展模式。

ANSI 标准要求将所有具有静态存储类型而又未显式初始化过的对象都置为零。c018.o/c018_e.o 和 c018i.o/c018i_e.o 启动代码模块都不满足这个要求。C18 提供了满足此要求的另一种启动模块 c018iz.o 和 c018iz_e.o。如果这种启动代码模块链接到了应用程序，那么除了初始化 idata 段外，所有具有静态存储类型而又未显式初始化过的对象都置为零。

为进行数据存储器的初始化， MPLINK 链接器在程序存储器中创建已初始化数据存储区的副本，启动代码把这个副本拷贝到数据存储器。 MPLINK 链接器会装入 .cinit 段以描述程序存储器映像应被拷贝到哪里。表 3-3 描述了 .cinit 段的格式。

表 3-3： .cinit 的格式

字段	描述	长度
num_init	段数	16 位
from_addr_0	段 0 的程序存储器起始地址	32 位
to_addr_0	段 0 的数据存储器起始地址	32 位
size_0	对于段 0，要初始化的数据存储器字节数	32 位
...	...	...
from_addr_*n*(1)	段 *n*(1) 的程序存储器起始地址	32 位
to_addr_*n*(1)	段 *n*(1) 的数据存储器起始地址	32 位
size_*n*(1)	对于段 *n*(1)，要初始化的数据存储器字节数	32 位

注 **1：** n = num_init - 1

启动代码设置了堆栈并选择性地拷贝了已初始化数据后，将调用 C 程序的 main() 函数。不向 main() 函数传递任何参数。MPLAB C18 通过循环调用将控制权转交给 main()，即：

```
loop:
  // Call the user's main routine
  main();
goto loop;
```

3.3.2 定制

要在器件 reset （复位）后，执行由编译器生成的任何其它代码之前，立即执行应用代码，则应编辑所需要的启动文件并把代码添加到 _entry() 函数的开头。

使用非扩展模式时定制启动文件：

1. 转至c:\mcc18\src\traditional\startup目录，其中c:\mcc18是安装编译器的目录。
2. 编辑 c018.c、 c018i.c 或 c018iz.c 添加需要的定制启动代码。
3. 编译更新了的启动文件生成 c018.o、 c018i.o 或 c018iz.o。
4. 将启动模块复制到 c:\mcc18\lib，其中 c:\mcc18 是安装编译器的目录。

使用扩展模式时定制启动文件：

1. 转至 c:\mcc18\src\extended\startup 目录，其中 c:\mcc18 是安装编译器的目录。
2. 编辑 c018_e.c、 c018i_e.c 或 c018iz_e.c 添加需要的定制启动代码。
3. 编译更新了的启动文件生成 c018_e.o、 c018i_e.o 或 c018iz_e.o。
4. 将启动模块复制到 c:\mcc18\lib，其中 C:\mcc18 是安装编译器的目录。

3.4 编译器管理的资源

PIC18 PICmicro 单片机的某些特殊功能寄存器和数据段被 MPLAB C18 使用，用户代码不能使用它们。表 3-4 列出了这些资源，以及这些资源对于编译器的主要用途，同时还列出了进入中断服务程序时编译器是否会自动保存这些资源。

表 **3-4**：　编译器保留的资源

编译器管理的资源	主要用途	自动保存
PC	执行控制	✓
WREG	中间计算	✓
STATUS	计算结果	✓
BSR	存储区选择	✓
PROD	乘法结果、返回值和中间计算	
section.tmpdata	中间计算	
FSR0	指向 RAM 的指针	✓
FSR1	栈指针	✓
FSR2	帧指针	✓
TBLPTR	存取程序存储器中的值	
TABLAT	存取程序存储器中的值	
PCLATH	函数指针调用	
PCLATU	函数指针调用	
section MATH_DATA	数学库函数的参数、返回值和临时地址	

注：　编译器临时变量存放在一个名为 .tmpdata 的 udata 段中。每个中断服务程序都为临时数据的存储创建一个独立的段（参见 **2.9.2 节 "#pragma interruptlow fname / #pragma interrupt fname"**）。

注：



第 4 章 优化

MPLAB C18 编译器是一种优化编译器，其进行优化的主要目的是简化代码，减少代码量。在默认情况下，启用可由 MPLAB C18 编译器执行的所有优化功能，但也可以使用-O- 命令行选项禁用所有优化功能。 MPLAB C18 编译器也允许逐项启用或禁用优化功能。表 4-1 概括了可由 MPLAB C18 编译器执行的每项优化功能，包括启用或禁用每个优化功能的命令行选项，每个优化功能是否会影响调试，以及每个优化功能将在哪个章节中论述。

注：	不会优化任何包含行内汇编代码的函数。

表 4-1： MPLAB C18 优化功能

优化	启用	禁用	是否影响调试	章节
合并相同的字符串	-Om+	-Om-		4.1
转移优化	-Ob+	-Ob-		4.2
存储区选择优化	-On+	-On-		4.3
W 寄存器内容跟踪	-Ow+	-Ow-		4.4
代码排序	-Os+	-Os-		4.5
尾部合并	-Ot+	-Ot-	✓	4.6
删除执行不到的代码	-Ou+	-Ou-	✓	4.7
复制传递	-Op+	-Op-	✓	4.8
冗余存储删除	-Or+	-Or-	✓	4.9
删除死代码	-Od+	-Od-	✓	4.10
过程抽象	-Opa+	-Opa-	✓	4.11

4.1 合并相同的字符串 -Om+ / -Om-

启用合并相同的字符串优化功能时，会将两个或更多个相同的字符串合并成一个字符串表入口，在程序存储器中只存储一份原始数据。例如，在如下语句中，启用合并相同的字符串功能 （-Om+）时，仅会在输出目标文件中存储字符串 “foo”数据的一个实例，而且 a 和 b 都引用这个数据。

```
const rom char * a = "foo";
const rom char * b = "foo";
```

-Om-命令行选项禁用合并相同的字符串。

合并相同的字符串不影响源代码的调试。

4.2 转移优化 -Ob+ / -Ob-

指定-Ob+ 命令行选项时， MPLAB C18 编译器进行以下转移优化：

1. 如果一个转移 （条件转移或无条件转移）跳转到另一个无条件转移，则可将前面的转移修改为直接跳转到后面无条件转移的目标语句行。
2. 跳转到 RETURN、 ADDULNK 或 SUBULNK 指令的无条件转移可分别用一条 RETURN、 ADDULNK 或 SUBULNK 指令替代。
3. 跳转到紧随其后的下一条指令的转移 （有条件转移或无条件转移），可以删除。
4. 如果条件转移 1 跳转到条件转移 2，且两个转移的条件相同，则将条件转移 1 修改为跳转到条件转移 2 的目标语句行。
5. 如果条件转移及其后紧跟着的无条件转移跳转到相同的目标语句行，则删除有条件转移 （也就是说，无条件转移就足够了）。

-Ob- 命令行选项禁用转移优化。

有些转移优化节省程序空间，而有些转移优化则可能发现执行不到的代码，这些代码可以通过 “删除执行不到的代码”优化功能删除 （详情请参阅 **4.7 节 “删除执行不到的代码”**）。转移优化不影响源代码的调试。

4.3 存储区选择优化 -On+ / -On-

当确定存储区选择寄存器已经包含正确的值时，存储区选择优化将删除 MOVLB 指令。例如下列的 C 源代码片段：

```
unsigned char a, b;
a = 5;
b = 5;
```

如果编译时禁用存储区选择优化 （-On-）， MPLAB C18 将在每次赋值前都加载存储区选择寄存器：

```
0x000000 MOVLB a
0x000002 MOVLW 0x5
0x000004 MOVWF a,0x1
0x000006 MOVLB b
0x000008 MOVWF b,0x1
```

启用存储区选择优化 （-On+）编译这段代码， MPLAB C18 可以确定存储区选择寄存器的值不会改变而删除第二条 MOVLB 指令：

```
0x000000 MOVLB a
0x000002 MOVLW 0x5
0x000004 MOVWF a,0x1
0x000006 MOVWF b,0x1
```

存储区选择优化不影响源代码的调试。

4.4 W 寄存器内容跟踪 -Ow+ / -Ow-

当确定工作寄存器已经包含正确的值时， W 寄存器内容跟踪优化将删除 MOVLW 指令。例如下列的 C 源代码片段：

unsigned char a, b;

```
a = 5;
b = 5;
```

如果编译时禁用 W 寄存器内容跟踪 （-Ow-）， MPLAB C18 将在每次赋值前都把值 5 加载到工作寄存器：

```
0x000000 MOVLW 0x5
0x000002 MOVWF a,0x1
0x000004 MOVLW 0x5
0x000006 MOVWF b,0x1
```

启用 W 寄存器内容跟踪 （-Ow+）编译这段代码时， MPLAB C18 可以确定此时 W 寄存器的值已经是 5 而删除第二条 MOVLW 指令：

```
0x000000 MOVLW 0x5
0x000002 MOVWF a,0x1
0x000004 MOVWF b,0x1
```

W 寄存器内容跟踪不影响源代码的调试。

4.5 代码排序 -Os+ / -Os-

代码排序优化对代码序列重新排序，使代码按执行顺序排列。这将移动或者删除转移指令，从而使代码更少，效率更高。例如下面的代码：

```
first:
  sub1();
  goto second;
third:
  sub3();
  goto fourth;
second:
  sub2();
  goto third;
fourth:
  sub4();
```

在此例中，按照数字编号顺序调用函数，即：先调用 sub1、 sub2、 sub3，然后调用 sub4。禁用代码排序优化 （-Os-）时，生成的汇编代码将反映出原来的程序流程：

```
0x000000 first  CALL sub1,0x0
0x000002
0x000004        BRA second
0x000006 third  CALL sub3,0x0
0x000008
0x00000a        BRA fourth
0x00000c second CALL sub2,0x0
0x00000e
0x000010        BRA third
0x000012 fourth CALL sub4,0x0
0x000014
```

启用代码排序优化 （-Os+）时，代码被重新排序，删除了转移指令：

```
0x000000 first  CALL sub1,0x0
0x000002
0x000004 second CALL sub2,0x0
0x000006
0x000008 third  CALL sub3,0x0
0x00000a
0x00000c fourth CALL sub4,0x0
0x00000e
```

代码排序优化不影响源代码的调试。

4.6 尾部合并 -Ot+ / -Ot-

尾部合并优化将多个相同的指令序列合并为一个指令序列。例如下面的 C 源代码片段：

```
if ( user_value )
  PORTB = 0x55;
else
  PORTB = 0x80
```

若编译时禁用尾部合并优化 （-Ot-），代码的 if 和 else 部分各生成一条 MOVWF PORTB,0x0 指令：

```
0x000000 MOVF user_value,0x0,0x0
0x000002 BZ 0xa
0x000004 MOVLW 0x55
0x000006 MOVWF PORTB,0x0
0x000008 BRA 0xe
0x00000a MOVLW 0x80
0x00000c MOVWF PORTB,0x0
0x00000e RETURN 0x0
```

然而，若编译时启用尾部合并优化 （-Ot+），那么代码的 if 和 else 部分仅生成和使用一条 MOVWF PORTB,0x0 指令：

```
0x000000 MOVF user_value,0x0,0x0
0x000002 BZ 0x8
0x000004 MOVLW 0x55
0x000006 BRA 0xa
0x000008 MOVLW 0x80
0x00000a MOVWF PORTB,0x0
0x00000c RETURN 0x0
```

若编译时启用尾部合并优化，则在调试源代码时，可能会加亮显示 “错误”的源代码行，因为两行或更多行源代码可能共用同一汇编代码序列，这使调试器很难确定正在执行哪一行源代码。

4.7 删除执行不到的代码 -Ou+ / -Ou-

删除执行不到的代码优化功能删除在正常的程序流程中不会执行的代码。例如下面的 C 源代码：

```
if (1)
{
  x = 5;
}
else
{
  x = 6;
}
```

显然，上面代码片段中的 `else` 部分绝对不可能执行到。如果禁用删除执行不到的代码优化 （`-Ou-`），生成的汇编代码将包含把 6 赋值给 x 所需的指令和有关的转移指令：

```
0x000000 MOVLB x
0x000002 MOVLW 0x5
0x000004 BRA 0xa
0x000006 MOVLB x
0x000008 MOVLW 0x6
0x00000a MOVWF x,0x1
```

如果启用删除执行不到的代码优化功能 （`-Ou+`），生成的汇编代码将不包含 else 部分的指令：

```
0x000000 MOVLB x
0x000002 MOVLW 0x5
0x000004 MOVWF x,0x1
```

删除执行不到的代码优化可能对在 C 源代码的某些行设置断点产生影响。

4.8 复制传递 -Op+ / -Op-

复制传递是这样一种变换，对于变量 x 和 y，进行 $x \leftarrow y$ 赋值后，那么在后面的代码中，只要中间插入的指令没有改变 *x* 和 *y* 的值，可用 *y* 代替 *x*。这种优化本身并不能节省指令，但是能实现删除死代码 （请参阅 **4.10 节 “删除死代码”**）。例如下面的 C 源代码：

```
char c;
void foo (char a)
{
  char b;
  b = a;
  c = b;
}
```

禁用复制传递 （`-Op-`），生成的汇编代码反映了原来的代码：

```
0x000000 foo    MOVFF a,b
0x000002
0x000004        MOVFF b,c
0x000006
0x000008        RETURN 0x0
```

启用复制传递 （-Op+），第二条指令将变成把 a 送到 c，而不是把 b 送到 c：

```
0x000000 foo    MOVFF a,b
0x000002
0x000004        MOVFF a,c
0x000006
0x000008        RETURN 0x0
```

然后，删除死代码优化将删除把 a 送到 b 这句无用的代码 （请参阅 **4.10 节 “删除死代码”**）。

复制传递可能会影响源代码的调试。

4.9 冗余存储删除 -Or+ / -Or-

如果指令序列中多次出现 $x \leftarrow y$ 的赋值，而其间的代码并没有改变 x 和 y 的值，则可以删除后面的赋值指令。这是公共子表达式删除的特例。例如下面的 C 源代码：

```
char c;
void foo (char a)
{
  c = a;
  c = a;
}
```

禁用冗余存储删除 （-Or-），生成的汇编代码将反映原来的代码：

```
0x000000 foo    MOVFF a,c
0x000002
0x000004        MOVFF a,c
0x000006
0x000008        RETURN 0x0
```

启用冗余存储删除 （-Or+），就不需要第二条把 c 赋值给 a 的指令了 。

```
0x000000 foo    MOVFF a,c
0x000002
0x000004        RETURN 0x0
```

冗余存储删除可能会对在 C 源代码的某些行设置断点产生影响。

4.10 删除死代码 -Od+ / -Od-

在函数中计算，但在至函数出口的任何路径上都没有使用过的值视为 “死” 值。只计算死值的指令视为 “死” 指令。函数作用域外的值视为使用过 （不是死值），因为不能确定这种值是否使用过。这里仍沿用 **4.8 节 “复制传递”** 中的例子：

```
char c;
void foo (char a)
{
      char b;
      b = a;
      c = b;
}
```

启用复制传递 （-Op+），禁用删除死代码 （-Od-），与 **4.8 节 “复制传递”** 中一样，生成的汇编代码如下：

```
0x000000 foo   MOVFF a,b
0x000002
0x000004       MOVFF a,c
0x000006
0x000008       RETURN 0x0
```

启用复制传递 （-Op+），并启用删除死代码 （-Od+），第二条指令将 a 赋值给 c，而不是将 b 赋值给 c，从而使对 b 的赋值成为死代码被删除：

```
0x000000 foo    MOVFF a,c
0x000002
0x000004        RETURN 0x0
```

删除死代码优化可能会对在 C 源代码的某些行设置断点产生影响。

4.11 过程抽象 -Opa+ / -Opa-

和大部分编译器一样， MPLAB C18 经常在一个目标文件中生成多次出现的代码序列。这项优化功能通过创建包含重复出现的代码的过程，并调用过程代替相同的代码序列，来减少生成的代码量。过程抽象对指定代码段中的所有函数进行优化。

注：	过程抽象节省了程序空间，但以牺牲执行时间为代价。

例如下面的 C 源代码片断：

```
distance -= time * speed;
position += time * speed;
```

若编译时禁用过程抽象（-Opa-），则为上面每条指令中的 time * speed 都生成了代码序列，如下面的黑体字所示：

```
0x000000 mainMOVLB time
0x000002  MOVF time,0x0,0x1
0x000004  MULWF speed,0x1
0x000006  MOVF PRODL,0x0,0x0
0x000008  MOVWF PRODL,0x0
0x00000a  CLRF PRODL+1,0x0
0x00000c  MOVF WREG,0x0,0x0
0x00000e  SUBWF distance,0x1,0x1
0x000010  MOVF PRODL+1,0x0,0x0
0x000012  SUBWFB distance+1,0x1,0x1
0x000014  MOVF time,0x0,0x1
0x000016  MULWF speed,0x1
0x000018  MOVF PRODL,0x0,0x0
0x00001a  MOVWF PRODL,0x0
0x00001c  CLRF PRODL+1,0x0
0x00001e  MOVF WREG,0x0,0x0
0x000020  ADDWF position,0x1,0x1
0x000022  MOVF PRODL+1,0x0,0x0
0x000024  ADDWFC position+1,0x1,0x1
0x000026  RETURN 0x0
```

然而，若编译时启用过程抽象（-Opa+），这两个代码序列就抽象为一个过程，通过调用过程来代替重复的代码。

```
0x000000 mainMOVLB time
0x000002  CALL _pa_0,0x0
0x000004
0x000006  SUBWF distance,0x1,0x1
0x000008  MOVF PRODL+1,0x0,0x0
0x00000a  SUBWFB distance+1,0x1,0x1
0x00000c  CALL _pa_0,0x0
0x00000e
0x000010  ADDWF position,0x1,0x1
0x000012  MOVF PRODL+1,0x0,0x0
0x000014  ADDWFC position+1,0x1,0x1
0x000016  RETURN 0x0
0x000018 __pa_0MOVF time,0x0,0x1
0x00001a  MULWF speed,0x1
0x00001c  MOVF PRODL,0x0,0x0
0x00001e  MOVWF PRODL,0x0
0x000020  CLRF PRODL+1,0x0
0x000022  MOVF WREG,0x0,0x0
0x000024  RETURN 0x0
```

仅执行一次过程抽象，并不能完成对所有匹配的代码序列的抽象。执行过程抽象，直到没有可以抽象的匹配代码序列或达到最大抽象次数（4 次）。可以通过-pa=*n* 命令行选项来控制过程抽象的次数。过程抽象可能会额外增加 2*n* - 1 级函数调用，其中 *n* 是总的过程抽象次数。如果应用中硬件堆栈资源有限，可以使用-pa=*n* 命令行选项来调整执行过程抽象的次数。

若编译时启用此优化功能，则在调试源代码时，可能会加亮显示某些 “错误的”源代码行，因为两行或更多代码行可能共用同一汇编代码序列，使调试器很难确定正在执行哪一行源代码。



第 5 章 示例应用程序

下面的示例应用程序将使得与 PIC18C452 单片机 PORTB 端口相连的 LED 闪烁。编译此应用程序的命令行是：

```
mcc18 -p 18c452 -I c:\mcc18\h leds.c
```

`c:\mcc18` 是安装编译器的目录。这个示例应用程序设计为与 PICDEM 2 演示板配合使用。示例程序包括：

1. 中断处理（`#pragma interruptlow`、中断向量、中断服务程序和现场保护）
2. 系统头文件
3. 针对处理器的头文件
4. `#pragma` *`sectiontype`*
5. 行内汇编

```
/*  1 */ #include <p18cxxx.h>
/*  2 */ #include <timers.h>
/*  3 */
/*  4 */ #define NUMBER_OF_LEDS 8
/*  5 */
/*  6 */ void timer_isr (void);
/*  7 */
/*  8 */ static unsigned char s_count = 0;
/*  9 */
/* 10 */ #pragma code low_vector=0x18
/* 11 */ void low_interrupt (void)
/* 12 */ {
/* 13 */   _asm GOTO timer_isr _endasm
/* 14 */ }
/* 15 */
/* 16 */ #pragma code
/* 17 */
/* 18 */ #pragma interruptlow timer_isr save=PROD
/* 19 */ void
/* 20 */ timer_isr (void)
/* 21 */ {
/* 22 */   static unsigned char led_display = 0;
/* 23 */
/* 24 */   INTCONbits.TMR0IF = 0;
/* 25 */
/* 26 */   s_count = s_count % (NUMBER_OF_LEDS + 1);
/* 27 */
/* 28 */   led_display = (1 << s_count++) - 1;
/* 29 */
/* 30 */   PORTB = led_display;
/* 31 */ }
/* 32 */
/* 33 */ void
/* 34 */ main (void)
/* 35 */ {
/* 36 */   TRISB = 0;
/* 37 */   PORTB = 0;
/* 38 */
/* 39 */   OpenTimer0 (TIMER_INT_ON & T0_SOURCE_INT & T0_16BIT);
/* 40 */   INTCONbits.GIE = 1;
/* 41 */
/* 42 */   while (1)
/* 43 */     {
/* 44 */     }
/* 45 */ }
```

第 1 行： 这一行包含了一般处理器头文件。通过命令行选项-p 选择正确的处理器（参阅 **2.5.1 节 “系统头文件”**和 **2.10 节 “针对处理器的头文件”**）。

第 10 行： 对于 PIC18 单片机来说，低优先级中断向量位于 000000018h。这行代码将当前代码段从默认代码段更改为地址为 0x18 处，名为 `low_vector` 的绝对代码段（参阅 **2.9.1 节 “#pragma sectiontype”**和 **2.9.2.3 节 “中断向量”**）。

第 13 行： 此行包含行内汇编，其功能是跳转到中断服务程序（参阅 **2.8.2 节 “行内汇编”**和 **2.9.2.3 节 “中断向量”**）。

第 16 行： 此行使编译器返回到默认的代码段（参阅 **2.9.1 节 “#pragma sectiontype”**和表 2-7）。

第 18 行： 此行将函数 `timer_isr` 声明为低优先级中断服务程序。为了使编译器为函数 `timer_isr` 生成 `RETFIE` 指令而不是 `RETURN` 指令，上述声明是必需的。另外，它还能确保特殊功能寄存器 `PROD` 的值得到保存。（**2.9.2 节 “#pragma interruptlow fname / #pragma interrupt fname”**和 **2.9.2.4 节 “中断现场保护”**）。

第 19–20 行： 这两行定义了函数 `timer_isr`。请注意：`timer_isr` 没有参数，也没有返回值（这是中断服务程序的要求）（请参阅 **2.9.2.2 节 “中断服务程序”**）。

第 24 行： 清除 `TMR0` 的中断标志，以避免程序多次处理同一个中断（参阅 **2.10 节 “针对处理器的头文件”**）。

第 30 行： 示范如何在 C 程序中修改特殊功能寄存器 `PORTB`（参阅 **2.10 节 “针对处理器的头文件”**）。

第 36-37 行： 初始化特殊功能寄存器 `TRISB` 和 `PORTB`（参阅 **2.10 节 “针对处理器的头文件”**）。

第 39 行： 允许 `TMR0` 中断，把定时器设置为内部 16 位时钟。

第 40 行： 允许全局中断（参阅 **2.10 节 “针对处理器的头文件”**）。

注：



附录 A COFF 文件格式

Microchip 的 COFF 规范基于 *Understanding and Using COFF*（Gintaras R. Gircys © 1988, O'Reilly and Associates, Inc）中描述的 UNIX® System V COFF 格式。本文还特别提到了 Microchip COFF 格式和 UNIX SystemV COFF 格式的不同之处。

A.1 struct filehdr — 文件头

filehdr 结构保存与文件有关的信息，是 COFF 文件的第一项，指示可选文件头、符号表和段头从何处开始。

```
typedef struct filehdr
{
   unsigned short f_magic;
   unsigned short f_nscns;
   unsigned long f_timdat;
   unsigned long f_symptr;
   unsigned long f_nsyms;
   unsigned short f_opthdr;
   unsigned short f_flags;
} filehdr_t;
```

A.1.1 unsigned short f_magic

标志数字 （magic number）用于识别文件遵循的 COFF 实现。对于 Microchip PICmicro 单片机 COFF 文件，标志数字是 0x1234。

A.1.2 unsigned short f_nscns

COFF 文件中的段数。

A.1.3 unsigned long f_timdat

COFF 文件创建时的时间和日期标记 （这个值是从 1970 年 1 月 1 日零时开始的秒数）。

A.1.4 unsigned long f_symptr

指向符号表的指针。

A.1.5 unsigned long f_nsyms

符号表中的记录数。

A.1.6 unsigned short f_opthdr

可选头记录的长度。

A.1.7 unsigned short f_flags

有关 COFF 文件所包含内容的信息。表 A-1 列出了不同的文件头标志，以及描述和各自的值。

表 A-1： 文件头标志

标志	描述	值
F_RELFLG	COFF 文件中已去掉重定位信息。	0x0001
F_EXEC	COFF 文件是可执行文件，没有未解析的外部符号。	0x0002
F_LNNO	COFF 文件中已去掉行号信息。	0x0004
L_SYMS	COFF 文件中已去掉局部变量。	0x0080
F_EXTENDED18	使用 扩展模式生成 COFF 文件。	0x4000
F_GENERIC	COFF 文件与处理器无关。	0x8000

A.2 struct opthdr — 可选文件头

opthdr 结构包含与实现有关的文件级信息。对于 Microchip PICmicro 单片机 COFF 文件， opthdr 结构用于指定目标处理器的型号、编译器 / 汇编器的版本以及定义重定位类型。

注意，可选文件头的格式是特定于实现的 （即 Microchip 可选文件头的格式和 System V 可选文件头的格式是不同的）。

```
typedef struct opthdr
{
   unsigned short magic;
   unsigned short vstamp;
   unsigned long proc_type;
   unsigned long rom_width_bits;
   unsigned long ram_width_bits;
} opthdr_t;
```

A.2.1 unsigned short magic

标志数字用于确定适当的格式。

A.2.2 unsigned short vstamp

版本标记。

A.2.3 unsigned long proc_type

目标处理器的类型。表 A-2 列出了处理器的类型和存储在此字段中的相应值。

表 **A-2**： 处理器类型

处理器	值
PIC18C242	0x8242
PIC18C252	0x8252
PIC18C442	0x8442
PIC18C452[(1)]	0x8452
PIC18C658	0x8658
PIC18C858	0x8858
PIC18C601	0x8601
PIC18C801	0x8801
PIC18F242	0x242F
PIC18F252	0x252F
PIC18F442	0x442F
PIC18F452	0x452F
PIC18F248	0x8248
PIC18F258	0x8258
PIC18F448	0x8448
PIC18F458	0x8458
PIC18F1220	0xA122
PIC18F1320	0xA132
PIC18F2220	0xA222
PIC18F2320	0xA232
PIC18F4220	0xA422
PIC18F4320	0xA432
PIC18F6520	0xA652
PIC18F6620	0xA662
PIC18F6720	0xA672
PIC18F8520	0xA852
PIC18F8620	0xA862
PIC18F8720	0xA872
PIC18F6585	0x6585
PIC18F6680	0x6680
PIC18F8585	0x8585
PIC18F8680	0x8680
PIC18F6525	0x6525
PIC18F6621	0xA621
PIC18F8525	0x8525
PIC18F8621	0x8621
PIC18F4331	0x4331
PIC18F4431	0x4431
PIC18F2331	0x2331
PIC18F2431	0x2431
PIC18F2439	0x2439
PIC18F2539	0x2539
PIC18F4439	0x4439
PIC18F4539	0x4539
PIC18F2585	0x2585
PIC18F2680	0x2680
PIC18F2681	0x2681
PIC18F4585	0x4585

表 A-2：　处理器类型 （续）

PIC18F4680	0x4680
PIC18F4681	0x4681
PIC18F2515	0x2515
PIC18F2525	0x2525
PIC18F2610	0x2610
PIC18F2620	0x2620
PIC18F4515	0x4515
PIC18F4525	0x4525
PIC18F4610	0x4610
PIC18F4620(2)	0x4620
PIC18F6410	0x6410
PIC18F6490	0x6490
PIC18F8410	0x8410
PIC18F8490	0x8490

注　**1：** 这是当编译器工作在非扩展模式时，为一般处理器编译时使用的处理器。
　　2： 这是当编译器工作在扩展模式时，为一般处理器编译时使用的处理器。

A.2.4 `unsigned long rom_width_bits`

程序存储器的宽度 （以 “位” 为单位）。

A.2.5 `unsigned long ram_width_bits`

数据存储器的宽度 （以 “位” 为单位）。

A.3 `struct scnhdr` — 段头

`scnhdr` 结构包含与某个段有关的信息。Microchip PIC 单片机 COFF 文件中的段名定义和段名的通常 COFF 定义略有不同。 Microchip PIC 单片机 COFF 段名的长度可以大于 8 个字符，Microchip PIC 单片机 COFF 文件允许为长的段名建立字符串表记录。

```
typedef struct scnhdr
{
   union
   {
     char _s_name[8] /* section name is a string */
     struct
       {
         unsigned long _s_zeroes
         unsigned long _s_offset
       }_s_s;
   }_s;

   unsigned long s_paddr;
   unsigned long s_vaddr;
   unsigned long s_size;
   unsigned long s_scnptr;
   unsigned long s_relptr;
   unsigned long s_lnnoptr;
   unsigned short s_nreloc;
   unsigned short s_nlnno;
   unsigned long s_flags;
} scnhdr_t;
```

A.3.1 union _s

字符串或者对字符串表的引用。少于 8 个字符的字符串直接存储，其它字符串存储到字符串表中。如果字符串的前 4 个字符是 0，那么最后 4 个字节作为在字符串表内的偏移量。这样做不太好，因为不严格符合 ANSI 的规定 （也就是说 type munging 不是标准定义的操作 ）。但这是有效的，它保持了和 System V 格式的二进制兼容性，而其它方式做不到这一点。这种实现的优点是可反映用于长符号名的标准 System V 结构。

A.3.1.1 char s_name[8]

直接存储的段名。如果段名少于 8 个字符，直接存储段名。

A.3.1.2 struct _s_s

段名存储在字符串表中。如果段名的前 4 个字符是 0，那么最后 4 个字节作为在字符串表内的偏移量，以找到段名。

A.3.1.3 unsigned long _s_zeroes

段名的前 4 个字符是 0。

A.3.1.4 unsigned long _s_offset

字符串表中段名的偏移量。

A.3.1.5 unsigned long s_paddr

段的物理地址。

A.3.1.6 unsigned long s_vaddr

段的虚拟地址。它总是和 s_paddr 包含同样的值。

A.3.2 unsigned long s_size

段的长度。

A.3.3 unsigned long s_scnptr

COFF 文件中此段的原始数据的指针。

A.3.4 unsigned long s_relptr

COFF 文件中此段的重定位信息的指针。

A.3.5 unsigned long s_lnnoptr

COFF 文件中此段的行号信息的指针。

A.3.6 unsigned short s_nreloc

此段的重定位记录数。

A.3.7 unsigned short s_nlnno

此段的行号记录数。

A.3.8 unsigned long s_flags

段类型和内容标志。定义段类型及段限定符的标志作为位域存储在 s_flags 字段中。为位域定义了掩码，以便于访问。表 A-3 列出了不同的段头标志，以及描述和各自的值。

表 A-3： 段头标志

标志	描述	值
STYP_TEXT	段包含可执行代码。	0x00020
STYP_DATA	段包含已初始化数据。	0x00040
STYP_BSS	段包含未初始化数据。	0x00080
STYP_DATA_ROM	段包含程序存储器的已初始化数据。	0x00100
STYP_ABS	段为绝对段。	0x01000
STYP_SHARED	段为各存储区所共享。	0x02000
STYP_OVERLAY	段和不同目标模块的其它同名段相共享相同的存储空间。	0x04000
STYP_ACCESS	段可使用存取位进行访问。	0x08000
STYP_ACTREC	段包含一个函数的重叠 （overlay）激活记录。	0x10000

A.4 struct reloc — 重定位记录

访问可重定位标识符 （变量、函数等）的任何指令都必须有重定位记录。和 System V 的重定位数据不同 （在 System V 中，偏移量存储到重定位到的地址中），加到符号基址上的偏移量存储在重定位记录中。这是必需的，因为 Microchip 的重定位不仅要把 “地址 + 偏移量” 的值填写到数据流中，而且会对代码作简单的修正。这是更直接的存储偏移量的方法，但会略微增加文件的大小。

```
typedef struct reloc
{
   unsigned long r_vaddr;
   unsigned long r_symndx;
   short r_offset;
   unsigned short r_type;
} reloc_t;
```

A.4.1 unsigned long r_vaddr

引用的地址 （相对于原始数据开头的字节偏移量）。

A.4.2 unsigned long r_symndx

符号表的索引。

A.4.3 short r_offset

加到符号 r_symndx 的地址上的有符号偏移量。

A.4.4 unsigned short r_type

重定位类型，实现定义的值。表 A-4 列出了重定位类型，以及描述和各自的值。

表 **A-4**： 重定位类型

类型	描述	值
RELOCT_CALL	CALL 指令 （对于 PIC18，仅指 CALL 指令的第一个字）	1
RELOCT_GOTO	GOTO 指令 （对于 PIC18，仅指 GOTO 指令的第一个字）	2
RELOCT_HIGH	地址的高 8 位	3
RELOCT_LOW	地址的低 8 位	4
RELOCT_P	PIC17 MOVFP 或MOVPF指令中P操作数的5位地址	5
RELOCT_BANKSEL	为符号生成适当的存储区切换指令	6
RELOCT_PAGESEL	为符号生成适当的页切换指令	7
RELOCT_ALL	地址的 16 位	8
RELOCT_IBANKSEL	生成间接的存储区选择指令	9
RELOCT_F	PIC17 MOVFP 或MOVPF指令中F操作数的8位地址	10
RELOCT_TRIS	TRIS 指令的数据寄存器地址	11
RELOCT_MOVLR	MOVLR — PIC17 的存储区选择指令	12
RELOCT_MOVLB	MOVLB — PIC17 和 PIC18 的存储区选择指令	13
RELOCT_GOTO2	PIC18 GOTO 指令的第二个字	14
RELOCT_CALL2	PIC18 CALL 指令的第二个字	14
RELOCT_FF1	PIC18 MOVFF 指令的源寄存器	15
RELOCT_FF2	PIC18 MOVFF 指令的目标寄存器	16
RELOCT_SF2	PIC18 MOVSF 指令的目标寄存器	16
RELOCT_LFSR1	PIC18 LFSR 指令的第一个字	17
RELOCT_LFSR2	PIC18 LFSR 指令的第二个字	18
RELOCT_BRA	PIC18 的 BRA 指令	19
RELOCT_RCALL	PIC18 的 RCALL 指令	19
RELOCT_CONDBRA	PIC18 的相对条件转移指令	20
RELOCT_UPPER	24 位地址的最高 8 位	21
RELOCT_ACCESS	PIC18 的存取位	22
RELOCT_PAGESEL_WREG	用 W 寄存器作为暂存来选择正确的页	23
RELOCT_PAGESEL_BITS	用位置位 / 清除指令来选择正确的页	24
RELOCT_SCNSZ_LOW RELOCT_SCNSZ_HIGH RELOCT_SCNSZ_UPPER	段的长度	25 26 27
RELOCT_SCNEND_LOW RELOCT_SCNEND_HIGH RELOCT_SCNEND_UPPER	段结束地址	28 29 30
RELOCT_SCNEND_LFSR1 RELOCT_SCNEND_LFSR2	LFSR 指令的段结束地址	31 32

A.5 struct syment — 符号表记录

为所有标识符、段、函数开头、函数结束、程序块开头和程序块结束创建符号。

```
#define SYMNMLEN 8
struct syment
{
   union
   {
     char _n_name[SYMNMLEN];
     struct
     {
       unsigned long _n_zeroes;
       unsigned long _n_offset;
     } _n_n;
     char *_n_nptr[2];
   } _n;

   unsigned long n_value;
   short n_scnum;
   unsigned short n_type;
   char n_sclass;
   char n_numaux;
}
```

A.5.1 union _n

符号名可作为字符串直接存储，或作为对字符串表的引用。少于 8 个字符的符号名直接存储，其它符号名存储在字符串表中。正是受到这种结构的启发，扩展了段的数据结构，允许段名存储在符号表中。

A.5.1.1 char _n_name [SYMNMLEN]

如果符号名少于 8 个字符，直接存储。

A.5.1.2 struct _n_n

符号名存储在字符串表中。如果符号名的前 4 个字符是 0，那么符号名的最后 4 个字符就构成在字符串表中的偏移量，以找到符号名。

A.5.1.3 unsigned long _n_zeros

符号名的前 4 个字符为 0。

A.5.1.4 unsigned long _n_offset

字符串表中符号名的偏移量。

A.5.1.5 char *_n_nptr

允许重叠。

A.5.2 unsigned long n_value

符号的值。一般来说，这是符号在其所在段中的地址。对于链接时间常量 （例如，Microchip 的符号 _stksize），这个值是一个常数值，而不是一个地址。对于链接器，这一般没有什么差别，只是在应用代码中的用法不同。

A.5.3 short n_scnum

引用符号所在段的段号。

A.5.4 unsigned short n_type

基本类型和派生类型。

A.5.4.1 符号类型

表 A-5 列出了基本类型，以及描述和各自的值。

表 A-5： 基本符号类型

类型	描述	值
T_NULL	空	0
T_VOID	无类型	1
T_CHAR	字符型	2
T_SHORT	短整型	3
T_INT	整型	4
T_LONG	长整型	5
T_FLOAT	浮点型	6
T_DOUBLE	双精度浮点型	7
T_STRUCT	结构	8
T_UNION	联合	9
T_ENUM	枚举	10
T_MOE	枚举成员	11
T_UCHAR	无符号字符型	12
T_USHORT	无符号短整型	13
T_UINT	无符号整型	14
T_ULONG	无符号长整型	15

A.5.4.2 派生类型

通过派生类型来处理指针、数组和函数。表 A-6 列出了派生类型及其描述和各自的值。

表 A-6： 派生类型

派生类型	描述	值
DT_NON	无派生类型	0
DT_PTR	指针	1
DT_FCN	函数	2
DT_ARY	数组	3

A.5.5 char n_sclass

符号的存储类别。表 A-7 列出了存储类别及其描述和各自的值。

表 A-7： 存储类别

存储类别	描述	值
C_EFCN	函数的物理结束地址	0xFF
C_NULL	空	0
C_AUTO	自动变量	1
C_EXT	外部符号	2
C_STAT	静态型	3
C_REG	寄存器变量	4
C_EXTDEF	外部定义	5
C_LABEL	标号	6
C_ULABEL	未定义标号	7
C_MOS	结构成员	8
C_ARG	函数参数	9
C_STRTAG	结构标记	10
C_MOU	联合成员	11
C_UNTAG	联合标记	12
C_TPDEF	类型定义	13
C_USTATIC	未定义的静态型	14
C_ENTAG	枚举标记	14
C_MOE	枚举成员	16
C_REGPARM	寄存器参数	17
C_FIELD	位域	18
C_AUTOARG	自动参数	19
C_LASTENT	假进入 （程序块的结束）	20
C_BLOCK	“bb” 或 “eb”	100
C_FCN	“bf” 或 “ef”	101
C_EOS	结构结束	102
C_FILE	文件名	103
C_LINE	重新格式化为符号表记录的行号	104
C_ALIAS	副本标记	105
C_HIDDEN	dmert 公共库中的外部符号	106
C_EOF	文件结束	107
C_LIST	绝对列表 （Absolute listing）打开或关闭	108
C_SECTION	段	109

A.5.6 char n_numaux

符号的附加记录数。

A.6 struct coff_lineno — 行号记录

任何一行可执行源代码在与其段相关联的行号表中都有一个 coff_lineno 记录。对于 Microchip PICmicro 单片机 COFF 文件，这意味着每条指令都会有一个 coff_lineno 记录，因为调试信息常用于通过绝对列表文件进行调试。读者应该注意，并不要求 COFF 文件对于每条指令都有这一记录，尽管通常每条指令都有这一记录。这是和 System V 格式的明显区别。

```
struct coff_lineno2
{
   unsigned long l_srcndx;
   unsigned short l_lnno;
   unsigned long l_paddr;
   unsigned short l_flags;
   unsigned long l_fcnndx;
} coff_lineno_t;
```

A.6.1 unsigned long l_srcndx

相关联源文件的符号表索引。

A.6.2 unsigned short l_lnno

行号。

A.6.3 unsigned long l_paddr

行号记录所对应代码的地址。

A.6.4 unsigned short l_flags

行号记录的标志位。表 A-8 列出了标志位及其描述和相应的值。

表 A-8： 行号记录标志位

标志位	描述	值
LINENO_HASFCN	如果 l_fcndx 有效，置位该标志位	0x01

A.6.5 unsigned long l_fcnndx

相关联函数 （如果有的话）的符号表索引。

A.7 struct aux_file — 源文件的附加符号表记录

```
typedef struct aux_file
{
   unsigned long x_offset;
   unsigned long x_incline;
   unsigned char x_flags;
   char _unused[9];
} aux_file_t;
```

A.7.1 unsigned long x_offset

文件名的字符串表偏移量。

A.7.2 unsigned long x_incline

包含此文件的代码行行号。如果为 0，则没有包含此文件。

A.7.3 unsigned char x_flags

.file 记录的标志位。表 A-9 列出了标志位及其描述和相应的值。

表 A-9： .file 记录的标志位

标志位	描述	值
X_FILE_DEBUG_ONLY	此 .file 记录仅用于调试	0x01

A.8 struct aux_scn — 段的附加符号表记录

```
typedef struct aux_scn
{
   unsigned long x_scnlen;
   unsigned short x_nreloc;
   unsigned short x_nlinno;
   char _unused[10];
} aux_scn_t;
```

A.8.1 unsigned long x_scnlen

段的长度。

A.8.2 unsigned short x_nreloc

重定位记录数。

A.8.3 unsigned short x_nlinno

行号数。

A.9 struct aux_tag — struct/union/enum 标记名的附加符号表记录

```
typedef struct aux_tag
{
   char _unused[6];
   unsigned short x_size;
   char _unused2[4];
   unsigned long x_endndx;
   char _unused3[2];
} aux_tag_t;
```

A.9.1 unsigned short x_size

结构、联合或枚举的大小。

A.9.2 unsigned long x_endndx

此结构、联合或枚举的下一记录的符号索引。

A.10 struct aux_eos — struct/union/enum 结束的附加符号表记录

```
typedef struct aux_eos
{
   unsigned long x_tagndx;
   char _unused[2];
   unsigned short x_size;
   char _unused2[10];
} aux_eos_t;
```

A.10.1 unsigned long x_tagndx

结构、联合或枚举标记的符号索引。

A.10.2 unsigned short x_size

结构、联合或枚举的大小。

A.11 struct aux_fcn — 函数名的附加符号表记录

```
typedef struct aux_fcn
{
   unsigned long x_tagndx;
   unsigned long x_size;
   unsigned long x_lnnoptr;
   unsigned long x_endndx;
   short x_actscnum;
} aux_fcn_t;
```

A.11.1 unsigned long x_tagndx

如果返回值基本类型为结构或联合，与返回值类型相关联的结构标记名或联合标记名的符号表索引。

A.11.2 unsigned long x_lnnoptr

指向此函数行号的数据指针。

A.11.3 unsigned long x_endndx

此函数的下一条记录的符号索引。

A.11.4 short x_actscnum

静态激活记录数据的段号。

A.12 struct aux_fcn_calls — 函数调用的附加符号表记录

```
typedef struct aux_fcn_calls
{
   unsigned long x_calleendx;
   unsigned long x_is_interrupt;
   char _unused[10];
} aux_fcn_calls_t;
```

A.12.1 unsigned long x_calleendx

被调用函数的符号索引。如果要调用高阶函数，置位 AUX_FCN_CALLS_HIGHERORDER。

```
#define AUX_FCN_CALLS_HIGHERORDER ((unsigned long)-1)
```

A.12.2 unsigned long x_is_interrupt

指定函数是否是中断服务程序，如果是，指定中断的优先级。

0：不是中断

1：低优先级中断

2：高优先级中断

A.13 struct aux_arr — 数组的附加符号表记录

```
#define X_DIMNUM 4
typedef struct aux_arr
{
   unsigned long x_tagndx;
   unsigned short x_lnno;
   unsigned short x_size;
   unsigned short x_dimen[X_DIMNUM];
} aux_arr_t;
```

A.13.1 unsigned long x_tagndx

如果基本类型为结构或联合，这是与数组元素类型相关联的结构或联合标记名的符号表索引。

A.13.2 unsigned short x_size

数组的大小。

A.13.3 unsigned short x_dimen[X_DIMNUM]

前四维的大小。

A.14 struct aux_eobf — 块或函数结尾的附加符号表记录

```
typedef struct aux_eobf
{
   char _unused[4];
   unsigned short x_lnno;
   char _unused2[12];
} aux_eobf_t;
```

A.14.1 unsigned short x_lnno

块 / 函数结尾处 C 源代码行的行号 （相对于块 / 函数的开头）。

A.15 struct aux_bobf — 块或函数开头的附加符号表记录

```
typedef struct aux_bobf
{
   char _unused[4];
   unsigned short x_lnno;
   char _unused2[6];
   unsigned long x_endndx;
   char _unused3[2];
} aux_bobf_t;
```

A.15.1 unsigned short x_lnno

块 / 函数开头的 C 源代码行行号，从块 / 函数的开头算起。

A.15.2 unsigned long x_endndx

此块 / 函数的下一记录的符号索引。

A.16 struct aux_var — struct/union/enum 类型变量的附加符号表记录

```
typedef struct aux_var
{
   unsigned long x_tagndx;
   char _unused[2];
   unsigned short x_size;
   char _unused2[10];
} aux_var_t;
```

A.16.1 unsigned long x_tagndx

结构、联合或枚举标记的符号索引。

A.16.2 unsigned short x_size

结构、联合或枚举的大小。

A.17 struct aux_field — 位域的附加记录

```
typedef struct aux_field
{
   char _unused[6];
   unsigned short x_size;
   char _unused2[10];
} aux_field_t;
```

A.17.1 unsigned short x_size

位域的长度，以 “位” 为单位。



附录 B 采用 ANSI 定义的方式

B.1 简介

本章讲述 MPLAB C18 采用 ANSI 定义的方式。 C 的 ISO 标准要求供应商提供 “采用定义的” 语言特点的具体细节方面的文档。

注：	括号中的章节号，例如 （6.1.2），参考 ANSI C 标准 X3.159-1989。

ANSI C 标准的 G.3 一章对以下各节列出的采用定义的方式都作了规定。

B.2 标识符

ANSI C 标准： 无外部链接的标识符中有效字符的数目 （超过 31）（6.1.2）

“有外部链接的标识符中有效字符的数目 （超过 6）（6.1.2）。”
“有外部链接的标识符中是否区分大小写 （6.1.2）。”

MPLAB C18 采用： 所有 MPLAB C18 标识符有至少 31 个有效字符。在有外部链接的标识符中区分大小写。

B.3 字符

ANSI C 标准： 包含不只一个字符的字符常量的值，或包含不只一个多字节字符的宽字符常量的值 （6.1.3.4）。”

MPLAB C18 采用： 整型字符常量的值是第一个字符的 8 位值。不支持宽字符。

ANSI C 标准： “不带符号说明的 `char` 和 `signed char` 还是 `unsigned char` 有相同的值域 （6.2.1.1）。”

MPLAB C18 采用： 不带符号说明的 `char` 和 `signed char` 有相同的值域。对于 MPLAB C18，可以通过命令行选项 （`-k`），将不带符号说明的 `char` 设为与 `unsigned char` 的值域相同。

B.4 整型

ANSI C 标准： “在使用 int 或 unsigned int 的表达式中 ，都可以使用 char、short int 或 int 位域，或这些类型的有符号或无符号变体，或枚举类型。如果 int 可以表示原始类型的所有值，那么把值转化为 int ；否则把值转化为 unsigned int。这叫做整型的提升。整型的提升不会改变所有其它算术类型。”

“整型的提升保持包括符号在内的值不变 （6.2.1.1）。”

MPLAB C18 采用： MPLAB C18 默认情况下不强制执行整型的提升，可以通过-Oi 选项要求编译器强制执行 ANSI 定义的执行方式。参见 **2.7.1 节 “整型的提升”**。

ANSI C 标准： “如果不能表示值，把一个整型转化为有符号的较短整型的结果，或者把无符号整型转化为等长的有符号整型的结果 （6.2.1.2）。”

MPLAB C18 采用： 当从较长的整型转化为较短的整型时，会舍弃值的高位，而按照较短的整型来转化剩下的位。从无符号整型转化为同长度的有符号整型时，按照同长度的有符号整型的规则来重新转化无符号整型的位。

ANSI C 标准： “对有符号整型进行位逻辑运算的结果 （6.3）”

MPLAB C18 采用： 进行位逻辑运算时，将有符号整型视为为同类型的无符号整型（也就是说，将符号位视为与任何其它位一样）。

ANSI C 标准： “整型除法中余数的符号 （6.3.5）。”

MPLAB C18 采用： 余数和商的符号相同。

ANSI C 标准： “负值有符号整型右移的结果 （6.3.7）。”

MPLAB C18 采用： 相当于把它作为相同长度的无符号整型进行右移 （也就是说，符号位并没有保留下来）。

B.5 浮点数

ANSI C 标准： “各种类型浮点数的表示和数值范围 （6.1.2.5）。”

“整数转化为浮点数，而此浮点数不能精确表示整数的值时的截取方向 （6.2.1.3）。”

“一个浮点数转化为较短的浮点数时的截取方向或者舍入方法（6.2.1.4）。”

MPLAB C18 采用： 参见 **2.1.2 节 “浮点型”**。

使用舍入到最接近值的方法。

B.6 数组和指针

ANSI C 标准： “保存数组最大长度所需的整型 — 即 sizeof 运算符的类型，size_t （6.3.3.4，7.1.1）。”

MPLAB C18 采用： 把 size_t 定义为 unsigned short long int。

ANSI C 标准： “把指针转换为整型或者反之的结果 （6.3.4）。”

MPLAB C18 采用： 整型包含用来表示指针的二进制值。如果指针长度大于整数长度，那么就会被截取以符合整型的长度。

ANSI C 标准： “保存指向同一数组的元素的两个指针之间的差别所需的整型，ptrdiff_t （6.3.6, 7.1.1）。”

MPLAB C18 采用： 把 ptrdiff_t 定义为 unsigned short long。

B.7 寄存器

ANSI C 标准： “通过 register 存储类别限定符指定把对象放入寄存器。（6.5.1）。”

MPLAB C18 采用： 忽略 register 存储类别限定符。

B.8 结构和联合

ANSI C 标准： “使用不同类型的成员访问联合对象的成员 （6.3.2.3）。”

MPLAB C18 采用： 成员的值是被转换为被访问成员类型的成员所在地址的位。

ANSI C 标准： “结构中成员的填充和对齐 （6.5.2.1）。”

MPLAB C18 采用： 结构和联合的成员按字节边界对齐。

B.9 位域

ANSI C 标准： 不带类型说明的 int 位域视为 signed int 位域还是 unsigned int 位域 （6.5.2.1）。

MPLAB C18 采用： 不带类型说明的 int 位域视为 signed int 位域。

ANSI C 标准： “一个单元内部位域的分配顺序 （6.5.2.1）。”

MPLAB C18 采用 : 位域按出现的顺序从最低有效位到最高有效位进行分配。

ANSI C 标准： “位域能否跨越存储单元的边界 （3.5.2.1）。”

MPLAB C18 采用： 位域不能跨越存储单元的边界。

B.10 枚举

ANSI C 标准： “表示枚举类型的值的整型 （6.5.2.2）。”

MPLAB C18 采用： 能表示枚举类型中所有值的最小类型。

B.11 Switch 语句

ANSI C 标准： “switch 语句中 case 分支的最大数目 （6.6.4.2）。”

MPLAB C18 采用： 分支的最大数目仅受目标存储器的限制。

B.12 预处理伪指令

ANSI C 标准： “定位可包含的源文件的方法 （6.8.2）。”

MPLAB C18 采用： 参见 **2.5.1** 节 **“系统头文件”**。

ANSI C 标准： “对可包含的源文件的引用名的支持 （6.8.2）。”

MPLAB C18 采用： 参见 **2.5.2** 节 **“用户头文件”**。

ANSI C 标准： “对每个识别到的 #pragma 伪指令的执行方式 （6.8.6）。”

MPLAB C18 采用： 参见 **2.9** 节 **“Pragma 伪指令”**。

附录 C 命令行概述

用法： `mcc18 [options] file [options]`

表 **C-1**： 命令行概述

选项	描述	参考
-?, --help	显示帮助界面	1.2.2
-I=<path>	添加文件包含路径 'path'	2.5.1， 2.5.2
-fo=<name>	目标文件名	1.2.1
-fe=<name>	错误文件名	1.2.1
-k	把无符号说明的字符型设置为无符号字符型	2.1
-ls	大型堆栈 （可以跨越多个存储区）	3.2.2
-ms	把编译器存储模型设置为小存储模型 （默认）	2.6，3.1
-ml	把编译器存储模型设置为大存储模型	2.6，3.1
-O，-O+	启用所有优化功能 （默认）	4
-O-	禁用所有优化功能	4
-Od+	启用死代码删除 （默认）	4.10
-Od-	禁用死代码删除	4.10
-Oi+	启用整型提升	2.7.1
-Oi-	禁用整型提升 （默认）	2.7.1
-Om+	启用合并相同的字符串 （默认）	4.1
-Om-	禁用合并相同的字符串	4.1
-On+	启用存储区选择优化 （默认）	4.3
-On-	禁用存储区选择优化	4.3
-Op+	启用复制传递 （默认）	4.8，4.10
-Op-	禁用复制传递	4.8， 4.10
-Or+	启用冗余存储删除功能 （默认）	4.9
-Or-	禁用冗余存储删除功能	4.9
-Ou+	启用删除执行不到的代码功能 （默认）	4.7
-Ou-	禁用删除执行不到的代码功能	4.7
-Os+	启用代码排序 （默认）	4.5
-Os-	禁用代码排序	4.5
-Ot+	启用尾部合并 （默认）	4.6
-Ot-	禁用尾部合并	4.6
-Ob+	启用转移优化 （默认）	4.2
-Ob-	禁用转移优化	4.2
-sca	启用默认局部变量为 auto 型 （默认设置）。仅对非扩展模式有效。	2.3
-scs	启用默认局部变量为 static 型。仅对非扩展模式有效。	2.3
-sco	启用默认局部变量为 overlay 型 （静态分配激活记录）。仅对非扩展模式有效。	2.3

表 **C-1**： 命令行概述 （续）

选项	描述	参考
-Oa+	启用默认数据位于存取存储区。仅对非扩展模式有效。	2.9.1.3
-Oa-	禁用默认数据位于存取存储区 （默认设置）。仅对非扩展模式有效。	2.9.1.3
-Ow+	启用 W 寄存器内容跟踪 （默认）	4.4
-Ow-	禁用 W 寄存器内容跟踪	4.4
-Opa+	启用过程抽象 （默认）	4.11
-Opa-	禁用过程抽象	4.11
-pa=<repeat count>	设置过程抽象重复次数 （默认为 4）	4.11
-p=<processor>	设定处理器 （默认为一般处理器）	1.2.4，2.10
-D<macro>[=text]	定义宏	1.2.3
-w={1|2|3}	设定警告等级 （默认为 2）	1.2.2
-nw=<n>	禁止消息 <n>	1.2.2
-verbose	详细的操作信息 （显示标题信息和其它信息）	1.2
--extended	生成 扩展模式代码。	1.2.5
--no-extended	生成 非扩展模式代码。	1.2.5
--help-message-list	显示一个所有诊断消息的列表	1.2.2
--help-message-all	显示所有诊断消息的帮助	1.2.2
--help-message=<n>	显示关于诊断号 <n> 的帮助	1.2.2



附录 D MPLAB C18 诊断

本附录列出了 MPLAB C18 编译器生成的错误、警告和消息。

D.1 错误

1002： syntax error, '%s' expected

没有为预处理器结构语法指定符号。这种错误的原因通常是拼写错误、遗漏了伪指令所需的操作数和括号不配对。

1013： error in pragma directive

MPLAB C18 在等待正在解析的 pragma 伪指令的结束，但是并没有发现新的命令行。这种错误可能是由 pragma 伪指令后的多余文本引起的。

1014： attribute mismatch in resumption of section '%s'

MPLAB C18 要求先前声明过的段的属性必须和当前 `#pragma` *`sectiontype`* 伪指令中指定的属性相匹配。如果当前的 `#pragma` *`sectiontype`* 伪指令多次指定 `overlay` 或 `access`，也会发生这种错误。

1016： integer constant expected for #line directive

#line 预处理伪指令中的行号操作数必须是整型常量。

1017： symbol name expected in 'interrupt' pragma

'save=' 子句后应指定一个用逗号分隔的静态分配作用域内符号名列表，这些符号名将由中断函数保存和恢复。通常导致这种错误的原因有：指定了不在作用域内的符号，没有包含定义所引用符号的头文件，以及输入了错误的符号名。

1018： function name expected in 'interrupt' pragma

要声明为中断服务程序的函数名应该作为 'interrupt' pragma 伪指令的第一个参数。函数符号必须在当前作用域内，并且没有参数和返回值。通常导致这种错误的原因是缺少被声明为中断服务程序的函数的原型和输入错误。

1019： '%s' is a compiler managed resource - it should not appear in a save= list

中断声明中 'save=' 子句中的符号名无效。如果用 'save=' 子句对某些地址进行保存 / 恢复会产生异常代码。不必对这些地址进行现场保护，只要从 'save=' 子句中删除这些地址就可以纠正此错误。

1020： unexpected input following '%s'

指定预处理器结构中有多余的信息。

1050： section address permitted only at definition

`#pragma` *`sectiontype`* 伪指令 *location* 子句中的绝对地址只能在定义此段的第一个 pragma 伪指令中指定。

1052： section overlay attribute does not match definition

MPLAB C18 要求先前定义的段属性必须与当前 `#pragma` *`sectiontype`* 伪指令中指定的属性相匹配。

1053： section share attribute does not match definition

MPLAB C18 要求先前定义的段属性必须与当前在 `#pragma` *`sectiontype`* 伪指令中指定的属性相匹配。

1054： section type does not match definition

MPLAB C18 在此前遇到过这个段名，但类型不同 （如 `code`、 `idata`、 `udata` 或 `romdata`）。

1099： %s

源代码 ‘#error’伪指令消息

1100： syntax error

无效的函数类型定义。

1101： lvalue required

需要一个指定对象的表达式。通常导致这种错误的原因包括遗漏了圆括号和遗漏了 ‘*’运算符。

1102： cannot assign to ‘const’ modified object

‘const’限定的对象定义为只读数据，因此不允许修改。

1103： unknown escape sequence ‘%s’

编译器不能识别指定的转义序列。可以查看 *MPASM™ User’s Guide with MPLINK™ Linker and MPLIB™ Librarian* (DS33014) 中有效字符转义序列的列表。

1104： division by zero in constant expression

编译器不能处理包含以零为除数 （或模数）的常数表达式。

1105： symbol ‘%s’ has not been defined

变量在被定义之前就被引用了。通常导致这种错误的原因包括符号名拼写错误、缺少定义变量的头文件或对变量的引用仅在内部作用域中有效。

1106： ‘%s’ is not a function

只有作为函数名的符号才能声明为中断函数。

1107： interrupt functions must not take parameters

处理器跳转到中断服务程序时不传递参数，因此，声明为中断函数的函数不能带有参数。

1108： interrupt functions must not return a value

中断函数是被处理器异步调用的，因此调用中断函数是没有返回值的。

1109： type mismatch in redeclaration of '%s'

声明的符号类型和此符号以前声明的类型不兼容。通常造成这种错误的原因包括缺少限定符或者限定符位置不对。

1110： 'auto' symbol '%s' not in function scope

自动变量只能分配到函数作用域内的堆栈中。

1111： undefined label '%s' in '%s'

标号由 'goto' 语句引用，但此标号并未在函数中定义。通常造成这种错误的原因包括标号标识符拼写错误，以及在标号的作用域外引用标号，即调用在其它函数中定义的标号。

1112： integer type expected in switch control expression

switch 语句的控制表达式必须是整型的。通常导致这种问题的原因包括遗漏了 '*' 运算符和 '[]' 运算符。

1113： integer constant expected for case label value

case 标号值必须是整型常量。

1114： case label outside switch statement detected

'case' 标号仅在 switch 语句内部有效。通常导致这种错误的原因是 '}' 位置不对。

1115： multiple default labels in switch statement

switch 语句只能有一个 'default' 标号。通常导致这种错误的原因是遗漏了 '}' 来结束内部的 switch。

1116： type mismatch in return statement

返回值的类型和声明的函数返回值类型不一致。通常导致这种错误的原因包括遗漏了 '*' 或者 '[]' 运算符。

1117： scalar type expected in 'if' statement

'if' 语句的控制表达式必须是标量类型，即整型或者指针。

1118： scalar type expected in 'while' statement

'while' 语句的控制表达式必须是标量类型，即整型或者指针。

1119： scalar type expected in 'do..while' statement

'do..while' 语句的控制表达式必须是标量类型，即整型或者指针。

1120： scalar type expected in 'for' statement

'for' 语句的控制表达式必须是标量类型，即整型或者指针。

1121： scalar type expected in '? :' expression

'?:' 运算符的控制表达式必须是标量类型，即整型或者指针。

1122： scalar operand expected for '!' operator

'!' 运算符的操作数必须是标量类型。

1123： scalar operands expected for '||' operator

逻辑 '或' 运算符 '||' 的操作数必须是标量类型。

1124： scalar operands expected for '&&' operator

逻辑 '与' 运算符 '&&' 的操作数必须是标量类型。

1125： 'break' must appear in a loop or switch statement

‘break’语句只能用于 ‘while’、‘do’、‘for’或 ‘switch’语句内。通常导致这种错误的原因是 ‘}’位置不对。

1126： 'continue' must appear in a loop statement

‘continue’语句只能用于 ‘while’、‘do’、‘for’或 ‘switch’语句内。

1127： operand type mismatch in '?:' operator

‘?:’运算符的结果操作数必须是标量类型或兼容的类型。

1128： compatible scalar operands required for comparison

比较运算符的操作数必须是兼容的标量类型。

1129： [] operator requires a pointer and an integer as operands

要求数组存取运算符 ‘[]’的一个操作数是指针，另一个操作数是整型。也就是说，对于 ‘x[y]’，表达式 ‘*(x+y)’一定是有效的。‘x[y]’在功能上等效于 ‘*(x+y)’。

1130: pointer operand required for '*' operator

反引用 （取值）运算符 ‘*’要求以指向非空对象的指针作为其操作数。

1131: type mismatch in assignment

赋值表达式运算符要求右侧表达式的结果与左侧表达式的结果类型兼容。通常导致这种错误的原因是遗漏了 ‘*’或者 ‘[]’运算符。

1132: integer type expected for right hand operand of '-=' operator

‘-=’运算符要求当运算符左侧为指针型时，右侧为整型。通常导致这种错误的原因是遗漏了 ‘*’或者 ‘[]’运算符。

1133: type mismatch in '-=' operator

‘-=’运算符的操作数类型要求：对于 ‘x-= y’，表达式 ‘x=x-y’有效。

1134: arithmetic operands required for multiplication operator

乘法运算符 ‘*’和 ‘*=’要求其操作数为算术类型。通常导致这种错误的原因是遗漏了反引用运算符 ‘*’或者下标运算符 ‘[]’。

1134: arithmetic operands required for division operator

除法运算符 ‘/’和 ‘/=’要求其操作数为算术类型。通常导致这种错误的原因是遗漏了反引用运算符 ‘*’或者下标运算符 ‘[]’。

1135: integer operands required for modulus operator

模运算符 ‘%’和 ‘%=’要求其操作数为算术类型。通常导致这种错误的原因是遗漏了反引用运算符 ‘*’或者下标运算符 ‘[]’。

1136: integer operands required for shift operator

移位运算符的操作数为整型。通常导致这种错误的原因是遗漏了反引用运算符 ‘*’或者下标运算符 ‘[]’。

1137: integer types required for bitwise AND operator

‘&’和‘&=’运算符要求两个操作数都为整型。通常导致这种错误的原因是遗漏了‘*’或者‘[]’运算符。

1138: integer types required for bitwise OR operator

‘|’和‘|=’运算符要求两个操作数都为整型。通常导致这种错误的原因是遗漏了‘*’或者‘[]’运算符。

1139: integer types required for bitwise XOR operator

‘^’和‘^=’运算符要求两个操作数都为整型。通常导致这种错误的原因是遗漏了‘*’或者‘[]’运算符。

1140: integer type required for bitwise NOT operator

‘~’运算符要求其操作数为整型。通常导致这种错误的原因是遗漏了‘*’或者‘[]’运算符。

1141: integer type expected for pointer addition

加法运算符要求当一个操作数为指针型时，另一个操作数必须为整型。通常导致这种错误的原因是遗漏了‘*’或者‘[]’运算符。

1142: type mismatch in ‘+’ operator

‘+’运算符的操作数类型必须符合以下条件：一个操作数为指针型，另一个操作数为整型；或者两个操作数都为算术型。

1143: pointer difference requires pointers to compatible types

当计算两个指针之间的差时，这两个指针必须指向类型兼容的两个对象。通常导致这种错误的原因是遗漏了括号和遗漏了‘[]’运算符。

1144: integer type required for pointer subtraction

如果减法运算符左侧的操作数为指针型，则右侧的操作数必须为整型。通常导致这种错误的原因是遗漏了‘*’或者‘[]’运算符。

1145: arithmetic type expected for subtraction operator

当减法运算符左侧的操作数不是指针型时，则要求减法运算符的两个操作数都为算术型。

1146: type mismatch in argument %d

函数调用的实参类型必须和相应形参所声明的类型兼容。

1147: scalar type expected for increment operator

递增运算符要求其操作数是标量类型的左值（lvalue），其值可以修改。

1148: scalar type expected for decrement operator

递减运算符要求其操作数是标量类型的左值（lvalue），其值可以修改。

1149: arithmetic type expected for unary plus

单目加运算符要求其操作数为算术型。

1150: arithmetic type expected for unary minus

单目减运算符要求其操作数为算术型。

1151: struct or union object designator expected

成员存取运算符 ‘.’ 和 ‘->’ 要求分别以结构 / 联合和结构 / 联合的指针作为操作数。

1152: scalar or void type expected for cast

显式强制类型转换要求其操作数类型为标量型，转换为标量型或 void 类型。

1153: cannot assign array type objects

不可以直接对数组类型的对象赋值，只可以对数组的成员赋值。

1154: parameter %d in ‘%s’ must have a name

定义函数时，必须对形参进行说明。形参说明要放在函数定义中，而不是函数原型中。

1160: conflicting storage classes specified

一个变量定义，只能给变量指定一个存储类别。

1161: conflicting base types specified

一个变量定义，只能给变量指定一个基本类型 （void、整型和浮点型等）。同一个基本类型的多次指定也是错误的 （例如 int int x;）。

1162: both ‘signed’ and ‘unsigned’ specified

一个类型只能包含一个 ‘signed’，或 ‘unsigned’，二者不能同时包含。

1163: function must be located in program memory

所有函数都必须位于程序存储器中，因为数据存储器是不可执行的。

1165: reference to incomplete tag ‘%s’

不能在声明中直接引用提前引用结构标记或联合标记。只能声明指向提前引用标记的指针。

1166: invalid type specification

类型说明无效。通常导致这种错误的原因包括输入错误或者误用了 typedef 类型。例如，‘int enum myEnum xyz;’ 有无效的类型说明。

1167: redefinition of enum tag ‘%s’

只能定义一次枚举标记。通常导致错误的原因是多次包含定义此枚举标记的头文件。

1168: reference to undefined enumeration tag ‘%s’

必须在引用枚举标记的任何声明前定义枚举标记。与结构标记和联合标记不同，提前引用枚举标记是不允许的。

1169: anonymous members allowed in unions only

匿名结构的成员只能定义为联合的成员。

1170: non-integral type bitfield detected

结构的位域成员的类型必须是整型。

1171: bitfield width greater than 8 detected

位域长度不能大于一个存储单元的长度，对于 MPLAB C18，一个存储单元是一个字节。因此，一个位域只能包含 8 位或更少位。

1172: enumeration value of ‘%s’ does not match previous

在多个枚举标记中使用相同的枚举常量名时，枚举常量的值在每个枚举中都必须相同。

1173: cannot locate a parameter in program memory, ‘%s’

因为所有的参数都位于堆栈中，在程序存储器中定位参数是不可能的。通常导致这种错误的原因是指向程序存储器的指针定义拼写错误。

1174: local ‘%s’ in program memory can not be ‘auto’

位于程序存储器中的局部变量必须声明为静态或者外部变量，因为 ‘auto’（自动）局部变量必须位于堆栈中。

1175: static parameter detected in function pointer ‘%s’

函数指针要求参数通过堆栈传递。当启用静态局部变量编译时，应将函数指针的参数以及函数指针所指向的函数的参数显式定义为 ‘auto’类型。

1176: the sign was already specified

一个类型要么是 ‘signed’型的，要么是 ‘unsigned’型的，只能指定为其中一种类型。

1200: cannot reference the address of a bitfield

不能直接引用结构的位域成员的地址。

1201: cannot dereference a pointer to ‘void’ type

‘*’ 反引用运算符要求一个指向非空对象的指针作为操作数。

1202: call of non-function

‘( )’ 函数调用后缀运算符的操作数必须是 “指向函数的指针”类型。通常这是一个函数标识符。通常导致这种错误的原因是遗漏了表示范围的括号。

1203: too few arguments in function call

调用函数时，传递的实参个数必须和函数定义中指定的形参个数相等。

1204: too many arguments in function call

调用函数时，传递的实参个数必须和函数定义中指定的形参个数相等。

1205: unknown member ‘%s’ in ‘%s’

结构标记或者联合标记中没有指定的成员名。通常导致这种错误的原因是成员名拼写错误和遗漏了嵌套结构中的成员访问运算符。

1206: unknown member ‘%s’

结构或者联合类型中没有指定的成员名。通常导致这种错误的原因是成员名拼写错误和遗漏了嵌套结构中的成员访问运算符。

1207: tag ‘%s’ is incomplete

成员访问运算符不能引用不完整的结构标记或联合标记。通常导致这种错误的原因是在符号定义时结构标记名拼写错误。

1208: "#pragma interrupt" detected inside function body

‘interrupt’ pragma 伪指令只能在文件级范围内使用。

1209: unknown function '%s' in #pragma interrupt

‘interrupt’ pragma 伪指令要求在遇到 pragma 伪指令时，被声明为中断服务程序的函数有一个有效的原型。

1210: unknown symbol '%s' in interrupt save list

‘interrupt’ pragma 伪指令要求列在 ‘save’列表中的符号必须已经声明过而且在作用域内。

1211: missing definition for interrupt function '%s'

函数被声明为中断服务程序，但是尚未定义。中断函数的定义必须和把函数声明为中断的 pragma 伪指令在同一个模块中。

1212: static function '%s' referenced but not defined

函数被声明为静态类型而且已经在模块内的其它地方被调用过，可是却没有定义。通常导致这种错误的原因是在函数定义中函数名拼写错误。

1213: initializer list expected

被初始化的符号需要一个包含在大括号内的初始化列表，但是却只有一个初始值。

1214: constant expression expected in initializer

静态分配的符号的初始值必须是常数表达式。

1215: initialization of bitfield members is not currently supported

目前不能进行位域结构成员的显式初始化。

1216: string initializer used for non-character array object

字符串初始化只有在初始化 ‘字符数组’或者 “字符指针”类型对象的时候才有效 （都可以是无符号字符型）。

1218: extraneous initializer values

初始化值的数目和被初始化对象类型要求的值数目不相同。在初始化列表中有太多值。

1219: integer constant expected

要求整型的常数表达式，但是却发现了非整型的表达式或者非常数表达式。

1220: initializer detected in typedef declaration of '%s'

typedef 定义中不能包括初始化。

1221: empty initializer list detected

初始化列表不能为空。在大括号中必须有一个或更多个初始值。

1250: '%s' operand %s must be a literal

操作码的此操作数必须为常数值，而不是符号引用。

1251: ‘%s’ operand count mismatch

此操作码的操作数数目和要求的数目不符。和 MPASM 不同， MPLAB C1x 行内汇编器要求显式说明所有的操作数。操作数 （例如存取位和目标位）没有默认值。

1252: invalid opcode ‘%s’ detected for processor ‘%s’

此操作码对目标处理器无效。通常导致这种错误的原因包括从不同指令集的处理器那里导入了行内汇编代码 （例如从 PIC17CXX 导入到 PIC18CXX）以及操作码拼写有误。

1253: constant operand expected

行内汇编操作码的操作数必须是一个常数表达式，这个常数表达式定义为常量或静态分配的符号引用，也可以对其加或减一个整型常量。通常导致这种错误的原因是将动态分配的符号 （auto 型局部变量和参数）用作行内汇编操作码的操作数。

1300: stack frame too large

栈帧的大小已经超出了最大的可寻址范围。通常导致这种错误的原因是在一个函数中把过多的局部变量分配为 auto 存储类型。

1301: parameter frame too large

参数帧的大小超出了最大可寻址范围。通常导致这种错误的原因是把太多参数传递到同一个函数。

1302: old style function declarations not supported

MPLAB C18 目前不支持旧的 K&R 类型的函数定义，而是采用 ANSI 标准推荐的嵌入参数类型声明。

1303: 'near’ symbol defined in non-access qualified section

分配到非存取段中的静态分配变量不能通过存取位访问，因此使用 near 范围限定符定义这些变量就会导致访问出错。

1304: illegal use of obsolete ‘overlay’ storage class for symbol ‘%s’

扩展模式下不支持 overlay 存储类别。还要注意在 非扩展模式下， overlay 存储类别仅对局部变量有效。

1500: unable to open file ‘%s’

编译器不能打开指定的文件。通常导致这种错误的原因包括文件名拼写错误和没有足够的访问权限。

1501: unable to locate file ‘%s’

编译器不能定位指定的文件。通常导致这种问题的原因包括文件名拼写错误和包含路径配置错误。

1502: unknown option ‘%s’
此命令行选项不是有效的 MPLAB C1X 选项。

1503: multi-bank stack supported only on 18Cxx core
只有 18CXX 处理器才支持软件堆栈跨越存储区边界。

1504: redefinition of ‘%s’
不能多次定义同一个函数名。

1505: redeclaration of label ‘%s’
不能多次定义同一个变量名。

1506: function ‘%s’ cannot have ‘overlay’ storage class specifier
不能对函数使用 overlay 存储类别说明符。

1507: variable ‘%s’ of ‘overlay’ storage class cannot have ‘near’ qualifier
编译器目前在存取 ram 中不支持 overlay 存储类别的变量。

1508: inconsistent linkage for %s
为标识符指定了内部和外部链接。

1509: %s cannot have ‘extern’ storage class
不能对参数使用 extern 存储类别说明符。

1510: %s cannot have ‘extern’ storage class, block scope, and an initializer
编译器不支持对 extern 存储类别的块作用域对象进行显式初始化。

1511: ran out of internal memory for temps
编译器不能再为任何临时变量分配存储空间。

1512: redefinition of label ‘%s’
不能在一个函数中多次定义同一个标号。

1513: redefinition of member ‘%s’
在结构或联合中，两个或两个以上成员不能使用相同的名字。

1514: cast of a pointer to floating point is undefined
请求的强制类型转换是非法的。这种错误可能是由赋值时遗漏了数组下标引起的。

1515: redefinition of case value %ld
switch 语句中，对于某个值只能有一个 case 分支。

1516: array size must be greater than zero
表示数组长度的常数值必须大于零。

1900: %s processor core not supported
编译器目前不支持指定的处理器内核。通常导致这种错误的原因是处理器名输入错误或者调用了错误的编译器可执行程序。

D.2 警告

2001: non-near symbol ‘%s’ declared in access section ‘%s’

声明到 access 段中的静态分配变量总是被链接器存放在存取数据存储区中，因此这些变量总是可以用 near 范围限定符限定。不指定 near 范围限定符也不会导致错误的代码，但可能产生额外的存储区选择指令。

2002: unknown pragma ‘%s’

编译器遇到不能识别的 pragma 伪指令。按照 ANSI/ISO 的要求，会忽略此 pragma 伪指令。通常导致这个警告的原因是 pragma 伪指令名拼写错误。

2025: default overlay locals is unsupported in Extended mode, -sco ignored

扩展模式下不支持 overlay 存储类别。

2026: default static locals is unsupported in Extended mode, -scs ignored

扩展模式下不支持将静态存储 （static）作为默认的存储类别。

2027: default auto locals is redundant in Extended mode, -sca ignored

扩展模式下，局部变量的默认存储类别始终为动态存储 （auto）。

2028: default static locals is unsupported in Extended mode, -Ol ignored

扩展模式下不支持将静态存储 （static）作为默认的存储类别。

2029: default access RAM is unsupported in Extended mode, -Oa ignored

静态模式下不支持 near 型变量的默认存取存储空间范围。

2052: unexpected return value

在定义为没有返回值的函数中发现返回值语句。这个返回值会被忽略。

2053: return value expected

定义为有返回值的函数没有返回值，返回值不确定。

2054: suspicious pointer conversion

在没有显式强制类型转换的情况下，将指针用作整型或者将整型用作指针。

2055: expression is always false

条件语句的控制表达式的值恒为 “假”。

2056: expression is always true

条件语句的控制表达式的值恒为 “真”。

2057: possibly incorrect test of assignment

对赋值表达式的隐式测试，（例如，经常看到 ‘if(x=y)’，但是其实想要使用的运算符是 ‘= =’，而不是 ‘=’）。

2058: call of function without prototype

调用函数时，在作用域内没有被调用函数的函数原型。这是不安全的，因为不能对函数实参进行类型检查。

2059: unary minus of unsigned value

单目减法运算符通常只用于有符号值。

2060: shift expression has no effect

移位的位数为 0 将不会改变被移位的数值。

2061: shift expression always zero

移位的位数比此被移位的数值的位数还要多，结果总是 0。

2062: '->' operator expected, not '.'

通过结构 / 联合的指针访问结构 / 联合的成员时，使用了 '.' 运算符。

2063: '.' operator expected, not '->'

用 "->" 运算符对结构 / 联合的成员进行了直接访问。

2064: static function '%s' not defined

函数被声明为静态类型，但是却没有对此函数进行定义。通常导致这种警告的原因是在函数定义时函数名输入错误。

2065: static function '%s' never referenced

定义过的静态函数未被调用。

2066: type qualifier mismatch in assignment

在指针赋值时，源指针和目的指针指向兼容类型的对象，但源指针指向以 'const' 或 'volatile' 限定的对象，目的指针却没有。

2067: type qualifier mismatch in argument %d

实参表达式是指向与形参类型兼容、以 'const' 或 'volatile 限定的对象的指针，但形参却是指向非 const 或 volatile 限定的对象的指针。

2068: obsolete use of implicit 'int' detected.

ANSI 标准允许声明变量时不指定基本类型，例如 "extern x;"，这里隐含了 'int' 基本类型。此标准没有采用这种做法，因此给出一个诊断信息。

2069: enumeration value exceeds maximum range

定义的枚举值，它超出了 signed long 格式的范围，而且枚举标记中有负的枚举值。此时将使用 unsigned long 类型来表示枚举，但不能正常进行包含负值的枚举常量的相对比较。

2070: constant value %d is too wide for bitfield and will be truncated

给定的值超出了位域的范围。用位域长度的 "与" 运算对给定值进行截取。

2071: %s cannot have 'overlay' storage class; replacing with 'static'

此时不允许 overlay 存储类别的参数。默认的局部变量存储类别是 overlay 时，应该指定参数为 static 存储类别。

2072: invalid storage class specifier for %s; ignoring

使用了不允许用于此定义的存储类型说明符。

2073: null-terminated initializer string too long

以零字符结尾的初始化字符串长度与数组对象的长度不符。

2100: obsolete use of ‘overlay’ for symbol ‘%s’, processing as auto

扩展模式下不支持 overlay 存储类别，将定义为此类别的变量按照 auto 存储类别处理。

2101: obsolete use of ‘static’ storage for parameter ‘%s’, treating as ‘auto’

在 扩展模式下编译时， MPLAB C18 要求所有函数参数为动态存储类别。更多信息，请参阅本用户指南的相关章节。

D.3 消息

3000: test of floating point for equality detected

测试两个浮点数是否相等有时可能得到错误的结果，因为由于舍入错误，两个算术上相等的表达式计算出的值可能会有很小的差异。

3001: optimization skipped for ‘%s’ due to inline assembly

无法对包含行内汇编的函数进行优化，因为行内汇编可能含有会导致优化器不能正确执行的结构。

3002: comparison of a signed integer to an unsigned integer detected

有符号值是负数时，把有符号整型值和无符号整型值放在一起比较可能会导致错误的结果。为比较无符号整型值和有符号值的二进制等价表示，有符号值应先被显式地转换为相同位数的无符号类型。

注：



附录 E　扩展模式

本附录详细描述 非扩展模式和扩展模式的差别。这些差别包括：

- 栈帧大小
- static 型参数
- overlay 关键字
- 行内汇编
- 预定义宏
- 命令行选项差别
- COFF 文件差别

E.1　源代码兼容性

E.1.1　栈帧大小

当编译器工作在扩展模式时，每个函数局部变量所占用的总存储空间大小限制为 96 字节；而工作在非扩展模式时，这个数据为 120 字节。

E.1.2　static 型参数

编译器工作在扩展模式时，不支持 static 型参数。编译器工作在扩展模式时，如果识别到 static 型参数，将发出警告诊断。在这种情况下，编译器将这种参数视为 auto 型，如同代码显式指定这种参数为 auto 型一样。因此参数将存储在堆栈中，而不是为其全局分配存储空间。由于每个函数 auto 型参数的总堆栈空间大小限制为 120 字节，将会发出应用程序 “parameter frame too large （参数帧太大）” 的诊断信息，而当编译器工作在非扩展模式时不会发生此问题。为解决这个问题，需要修改函数以减少其参数。

E.1.3　overlay 关键字

编译器工作在扩展模式时不支持 overlay 关键字。编译器工作在扩展模式时，如果识别到 overlay 关键字，将发出警告诊断。在这种情况下，编译器将这种参数视为 auto 型，如同代码显式指定这种参数为 auto 型一样。与 static 型参数类似，此时 overlay 局部变量将存储在堆栈中，而不是为其全局分配存储空间。由于每个函数局部变量的总存储空间限制为 96 字节，将会发出应用程序 “parameter frame too large （参数帧太大）” 的诊断信息，而当编译器工作在非扩展模式时不会发生此问题。为解决这个问题，需要修改函数以减少其 auto 型局部变量的个数。一种方法是将 overlay 变量更改为 static 型。

注：	不管编译器工作在何种模式下，均支持 Overlay 段 （#pragma overlay）。

E.1.4 行内汇编

工作在扩展模式时，编译器将在行内汇编中接受扩展指令 — ADDFSR、ADDULNK、CALLW、MOVSF、MOVSS、PUSHL、SUBFSR 和 SUBULNK；但工作在非扩展模式时，编译器在行内汇编中遇到扩展指令时将发出错误。

此外，工作在扩展模式时，编译器不会识别 MPASM 汇编器所使用、用来表示立即数变址寻址的方括号语法（如 CLRF [2]）。而是当 f 操作数小于或等于 0x5F 以及存取位操作数（a）置为 0 时（如 CLRF 2, 0），编译器将在行内汇编中识别立即数变址寻址。当编译器工作在非扩展模式时，这一相同的指令将解释为调用存取 RAM 。

E.1.5 预定义宏

无论编译器工作在何种模式，都可以在源代码中使用预定义宏来提高源代码的兼容性。当为扩展模式编译时，__EXTENDED18__ 预定义宏将为常数值 1；而当为非扩展模式编译时，__TRADITIONAL18__ 预定义宏将为常数值 1。

下面给出了几个使用预定义宏的具体例子：

1. 利用预定义宏来在非扩展模式下使用 static 型参数，而在扩展模式下使用 auto 型参数：

```
#ifdef __EXTENDED18__
   #define SCLASS auto
#else
   #define SCLASS static
#endif

void foo (SCLASS int bar);
```

2. 利用预定义宏来在非扩展模式下使用 overlay 关键字，而在扩展模式下使用 auto 关键字：

```
#ifdef __EXTENDED18__
   #define SCLASS auto
#else
   #define SCLASS overlay
#endif

void foo (void)
{
   SCLASS int bar;

   ...
}
```

3. 利用预定义宏来在非扩展模式下，在行内汇编中仅使用非扩展模式指令，而在扩展模式下，在行内汇编中使用扩展模式指令：

```
_asm
#ifdef __EXTENDED18__
   PUSHL 5
#else
   MOVLW 5
   MOVWF POSTINC1, 0
#endif
   ...
   MOVF POSTDEC1, 1, 0
_endasm
```


E.2 命令行选项差别

编译器工作在扩展模式时不支持下列命令行选项：

- 默认局部变量存储类别 （`-scs/-sco/-sca`）
  当编译器工作在扩展模式时，编译器仅支持默认 `auto` 局部变量。
- 默认数据存储在存取存储区 （`-Oa+/-Oa-`）
  由于工作在扩展模式时器件的存取 RAM 大小受限，因此工作在扩展模式时编译器不支持默认将数据存储在存取 RAM。

E.3 COFF 文件差别

E.3.1 一般处理器

当为一般处理器 （`-p18cxx`）编译时，在 COFF 文件的可选文件头中指定的处理器类型 （`proc_type`），编译器工作在扩展模式时将设置为 PIC18F4620，而编译器工作在非扩展模式时将设置为 PIC18C452。

E.3.2 文件头的 `f_flags` 字段

当工作在扩展模式时，生成的 COFF 文件将使文件头 `f_flags` 字段的 `F_EXTENDED18` 位置位。编译器工作在 非扩展模式时，此位不会被置位。

注：



术语表

A

ANSI

美国国家标准学会

B

八进制 （Octal）

使用数字 0-7，以 8 为基数的计数体制。最右边的位表示 1 的倍数，右侧第二位表示 8 的倍数，右侧第三位表示 82 = 64 的倍数，以此类推。

编译器 （Compiler）

将用高级语言编写的源文件翻译成机器代码的程序。

C

CPU

中央处理单元

存储类别 （Storage Class）

确定与指定对象相关联存储区的生存时间。

存储模型 （Memory Model）

一种描述，它指定指向程序存储器的指针的位数。

存储限定符 （Storage Qualifier）

表明所定义的对象的特性 （例如 CONST）。

存取存储区 （Access Memory）

PIC18 PICMICRO 单片机的一些特殊通用寄存器，对这些寄存器的访问与存储区选择寄存器 （BSR）的设置无关。

错误文件 （Error File）

包含 MPLAB C18 所生成的诊断信息的文件。

D

单片机 （Microcontroller）

高度集成的芯片，它包括 CPU、 RAM、某种类型的 ROM、 I/O 端口和定时器。

递归函数 （Recursive）

自调用的函数 （即调用自己的函数）。*参见递归 （RECURSIVE）*。

地址 （Address）

确定信息在存储器中位置的代码。

低字节低地址 （Little Endian）

将给定对象的最低有效字节存储在较低的地址。

段 （Section）

位于存储器特定地址的应用程序的一部分。

段属性 （Section Attribute）

段的特性 （如 ACCESS 段）。

E

二进制 （Binary）

使用数字 0 和 1，以 2 为基数的计数体制。最右边的位表示 1 的倍数，右边第二位表示 2 的倍数，右边第三位表示 22 = 4 的倍数，以此类推。

F

Free-standing

一种实现，它接受任何不使用复杂数据类型的严格符合程序，而且在这种实现中，对库条款中规定的属性的使用，仅限于标准头文件：<FLOAT.H>、 <ISO646.H>、<LIMITS.H>、<STDARG.H>、 <STDBOOL.H>、<stddef.h> 和 <stdint.h>。

非扩展模式 （Non-extended Mode）

在非扩展模式下，编译器不会使用扩展指令和立即数变址寻址。

G

高级语言 （High-level Language）

编写程序的语言，与汇编语言相比，它不依赖于具体的处理器。

H

汇编器 （Assembler ）

把汇编源代码翻译成机器代码的语言工具。

汇编语言 （Assembly）

以可读形式描述二进制机器代码的符号语言。

I

ICD

在线调试器

ICE

在线仿真器

IDE

集成开发环境

IEEE

电子和电气工程师协会

ISO

国际标准化组织

ISR

中断服务程序

J

绝对段 （Absolute Section）

具有链接器不能改变的固定地址的段。

K

可重定位 （Relocatable）

没有被指定到固定的存储器地址的对象。

可重入函数 （Reentrant）

可以有多个同时运行的实例的函数。在下面两种情况下可能发生函数重入：直接或间接递归调用函数；或者在由函数转入的中断处理过程中又执行此函数。

库 （Library）

可重定位目标模块的集合。

库管理器 （Librarian）

建立并管理库的程序。

扩展模式 （Extended Mode）

在扩展模式下，编译器将使用扩展指令 （即 `ADDFSR`、 `ADDULNK`、 `CALLW`、 `MOVSF`、 `MOVSS`、 `PUSHL`、 `SUBFSR` 和 `SUBULNK`）以及立即数变址寻址。

L

链接器 （Linker）

把目标文件和库文件结合起来生成可执行代码的程序。

M

MPASM 汇编器 （MPASM Assembler）

MICROCHIP PICMICRO 系列单片机的可重定位宏汇编器。

MPLIB 目标库管理器 （MPLIB Object Librarian ）

MICROCHIP PICMICRO 系列单片机的库管理器。

MPLINK 目标链接器 （MPLINK Object Linker）

MICROCHIP PICMICRO 系列单片机的链接器。

目标代码 （Object Code）

由汇编器或编译器生成的机器代码。

目标文件 （Object File）

包含目标代码的文件。它可以直接执行或需要与其它目标代码文件 （比如库文件）链接，以生成完全可执行的程序。

N

匿名结构 （Anonymous Structure）

未命名的对象。

P

Pragma

一种伪指令，它对于特定的编译器有意义。

R

RAM

随机访问存储器

ROM

只读存储器

S

十六进制 （Hexadecimal）

使用数字 0-9 以及字母 A-F （或 a-f），以 16 为基数的计数体制。字母 A-F 表示十进制数 10 到 15。最右边的位表示 1 的倍数，右边第二位表示 16 的倍数，第三位表示 162 = 256 的倍数，以此类推。

随机访问存储器 （Random Access Memory）

一种存储器，可以在这种存储器中以任意顺序读写信息。

T

特殊功能寄存器 （Special Function Register）

控制 I/O 处理函数、I/O 状态、定时器、其它模式或外围模块的寄存器。

条件编译 （Conditional Compilation）

只有当预处理伪指令指定的某个常数表达式为真时才编译程序段的操作。

X

向量 （Vector）

复位或中断发生时，应用程序跳转到的存储器地址。

Y

已分配段 （Assigned Section）

在链接器命令文件中已分配到目标存储区的段。

异步 （Asynchronously）

不同时发生的多个事件。通常用来指可能在处理器执行过程中的任意时刻发生的中断。

运行时模型 （Runtime Model）

编译器运行所遵循的各项前提。

Z

帧指针 （Frame Pointer）

指向堆栈中地址的指针，它用于区分堆栈中的函数参数和局部变量。

只读存储器 （Read Only Memory）

存储器硬件，它允许快速访问其中永久存储的数据，但不允许添加或更改数据。

致命错误 （Fatal Error）

引起编译立即停止的错误。不产生其它消息。

中断 （Interrupt）

发送到 CPU 的信号，它使 CPU 暂停正在运行的应用程序，把控制权转交给中断服务程序，以处理事件。执行完中断服务程序后，继续正常执行应用程序。

中断服务程序 （Interrupt Service Routine）

处理中断的函数。

中断响应时间 （Latency）

从事件发生到得到响应的时间。

中央处理单元 （Central Processing Unit）

芯片的一部分，其功能是取出要执行的指令，再对指令进行译码，然后执行指令。如果有必要，它和算术逻辑单元 （ ARITHMETIC LOGIC UNIT， ALU）一起工作，来完成指令的执行。它控制程序存储器的地址总线、数据存储器的地址总线和对堆栈的访问。

字节存储顺序 （Endianness）

多字节对象的字节存储顺序。

注：



索引

J

K

L

M

N

O

P

注：

注：



全球销售及服务网点

美洲

公司总部 Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 1-480-792-7200
Fax: 1-480-792-7277
技术支持：
http://support.microchip.com
网址：www.microchip.com

亚特兰大 Atlanta
Alpharetta, GA
Tel: 1-770-640-0034
Fax: 1-770-640-0307

波士顿 Boston
Westford, MA
Tel: 1-978-692-3848
Fax: 1-978-692-3821

芝加哥 Chicago
Itasca, IL
Tel: 1-630-285-0071
Fax: 1-630-285-0075

达拉斯 Dallas
Addison, TX
Tel: 1-972-818-7423
Fax: 1-972-818-2924

底特律 Detroit
Farmington Hills, MI
Tel: 1-248-538-2250
Fax: 1-248-538-2260

科科莫 Kokomo
Kokomo, IN
Tel: 1-765-864-8360
Fax: 1-765-864-8387

洛杉矶 Los Angeles
Mission Viejo, CA
Tel: 1-949-462-9523
Fax: 1-949-462-9608

圣何塞 San Jose
Mountain View, CA
Tel: 1-650-215-1444
Fax: 1-650-961-0286

加拿大多伦多 Toronto
Mississauga, Ontario,
Canada
Tel: 1-905-673-0699
Fax: 1-905-673-6509

亚太地区

中国 − 北京
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

中国 − 成都
Tel: 86-28-8676-6200
Fax: 86-28-8676-6599

中国 − 福州
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

中国 − 香港特别行政区
Tel: 852-2401-1200
Fax: 852-2401-3431

中国 − 上海
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

中国 − 沈阳
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

中国 − 深圳
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

中国 − 顺德
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

中国 − 青岛
Tel: 86-532-502-7355
Fax: 86-532-502-7205

台湾地区 − 高雄
Tel: 886-7-536-4818
Fax: 886-7-536-4803

台湾地区 − 台北
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

台湾地区 − 新竹
Tel: 886-3-572-9526
Fax: 886-3-572-6459

亚太地区

澳大利亚 Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

印度 India - Bangalore
Tel: 91-80-2229-0061
Fax: 91-80-2229-0062

印度 India - New Delhi
Tel: 91-11-5160-8632
Fax: 91-11-5160-8632

日本 Japan - Kanagawa
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

韩国 Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 或
82-2-558-5934

新加坡 Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

欧洲

奥地利 Austria - Weis
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

丹麦 Denmark - Ballerup
Tel: 45-4420-9895
Fax: 45-4420-9910

法国 France - Massy
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

德国 Germany - Ismaning
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

意大利 Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

荷兰 Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

英国 England - Berkshire
Tel: 44-118-921-5869
Fax: 44-118-921-5820

1020/04